修士 (工学) 学位論文

Master's Thesis of Engineering

メタ計算を用いた Continuation based C の検証手法

Verification Methods of Continuation based C using Meta Computations

2017 年 3 月

March 2017

比嘉 健太

Yasutaka HIGA

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

指導教員 : 教授　和田 知久

Supervisor: Prof. Tomohisa WADA

本論文は、修士 (工学) の学位論文として適切であると認める。

論　文　審　査　会

印

(主　査)　　和田　知久

印

(副　査)　　高良　富夫

印

(副　査)　　長田　智和

印

(副　査)　　河野　真治

要 旨

ソフトウェアが期待される仕様を満たすか検査することは重要である。特に実際に動作するソフトウェアを検証できるとなお良い。

本論文では Continuation based C(CbC) 言語で記述されたプログラムを検証用に変更せず信頼性を確保する手法を二つ提案する。一つはプログラムが持つ状態を数え挙げ、常に仕様を満たすことを保証するモデル検査的手法である。プログラムの実行を網羅的に行なうよう変更するメタ計算ライブラリ akasha を用いて赤黒木の仕様を検証する。

もう一つの信頼性向上手法としてデータ構造の持つ性質を証明する手法を提案する。プログラムにおける証明は Curry-Howard Isomorphism により型付き λ 計算に対応する。プログラムを型付けできるよう、CbC の型システムを部分型を用いて定義する。加えて、型の定義を用いて証明支援系言語 Agda 上で CbC のプログラムを記述し、データ構造の性質を証明する。

Abstract

Checking desirable specifications of software are important. If it checks actual implementations, much better.

In this papaer, We propose two verification methods using meta computations which save original implementations. On the hand method verify specification by enumerate possible states on programs. We checked red-black tree specification using our meta computation library named Akasha, which override program executions exhaustively.

On the other hand method veriy programs with proofs. Proposition and proofs have isomorphic relation to typed λ calculus by Curry-Howard Isomorphism. We define the CbC type system with subtype for proving CbC itself. Agda proves properties of translated CbC programs using proposed subtype definition.

目 次

図 目 次

表 目 次

リスト目次

第1章　Continuation based C

Continuation based C (CbC) は当研究室で開発しているプログラミング言語であり、OS や組み込みソフトウェアの開発を主な対象としている。CbC は C 言語の下位の言語であり、構文はほぼ C 言語と同じものを持つが、よりアセンブラに近い形でプログラムを記述する。CbC は CodeSegment と呼ばれる単位で処理を定義し、それらを組み合わせることにでプログラム全体を構成する。データの単位は DataSegment と呼ばれる単位で定義し、それら CodeSegment によって変更していくことでプログラムの実行となる。CbC の処理系には llvm/clang による実装 [1] と gcc [2] による実装などが存在する。

1.1　CodeSegment と DataSegment

本研究室では検証を行ないやすいプログラムの単位として CodeSegment と DataSegment を用いるプラグラミングスタイルを提案している。

CodeSegment は処理の単位である。入力を受け取り、それに対して処理を行なった後を出力を行なう。また、CodeSegment は他の CodeSegment と組み合わせることが可能である。ある CodeSegment A を CodeSegment B に接続した場合、 A の出力は B の入力となる。

DataSegment は CodeSegment が扱うデータの単位であり、処理に必要なデータが全て入っている。CodeSegment の入力となる DataSegment は Input DataSegment と呼ばれ、出力は Output DataSegment と呼ばれる。CodeSegment A と CodeSegment B を接続した時、A の Output DataSegment は B の入力 Input DataSegment となる。

1.2　Continuation based C における CodeSegment と DataSegment

最も基本的な CbC のソースコードをリスト 1.1 に、ソースコードが実行される流れを図 1.1 に示す。Continuation based C における CodeSegment は返り値を持たない関数として表現される。CodeSegment を定義するためには、C 言語の関数を定義する構文の返り値の型部分に `__code` キーワードを指定する。Input DataSegment は関数の引数として定

義される。次の CodeSegment へ処理を移す際には goto キーワードの後に CodeSegment 名と Input DataSegment を指定する。処理の移動を軽量継続と呼び、リスト 1.1 内の goto cs1(a+b); がこれにあたる。この時の (a+b) が次の CodeSegment である cs1 の Input DataSegment となる cs0 の Output DataSegment である。

リスト 1.1: CodeSegment の軽量継続

```
1 __code cs0(int a, int b){
2   goto cs1(a+b);
3 }
4
5 __code cs1(int c){
6   goto cs2(c);
7 }
```

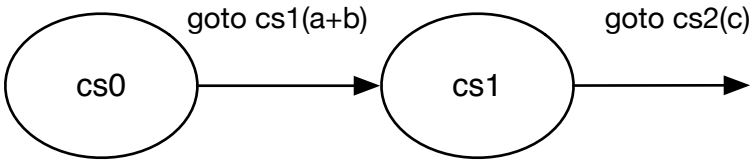

図 1.1: CodeSegment の軽量継続

Scheme などの call/cc といった継続はトップレベルから現在までの位置を環境として保持する。通常環境とは関数の呼び出しスタックの状態である。CbC の軽量継続は呼び出し元の情報を持たないため、スタックを破棄しながら処理を続けていく。よって、リスト 1.1 のプログラムでは cs0 から cs1 へと継続した後に cs0 へ戻ることはできない。

もう少し複雑な CbC のソースコードをリスト 1.2 に、実行される流れを図 1.2 に示す。このソースコードは整数の階乗を求めるプログラムである。CodeSegment factorial0 では自分自身への再帰的な継続を用いて階乗を計算している。軽量継続時には関数呼び出しのスタックは存在しないが、計算中の値を DataSegment で持つことで再帰を含むループ処理も行なうことができる。

リスト 1.2: 階乗を求める CbC プログラム

```
1 __code print_factorial(int prod)
2 {
3   printf("factorial = %d\n", prod);
4   exit(0);
5 }
6
7 __code factorial0(int prod, int x)
8 {
```

```
9   if (x >= 1) {
10    goto factorial0(prod*x, x-1);
11  } else {
12    goto print_factorial(prod);
13  }
14
15 }
16
17 __code factorial(int x)
18 {
19   goto factorial0(1, x);
20 }
21
22 int main(int argc, char **argv)
23 {
24   int i;
25   i = atoi(argv[1]);
26
27   goto factorial(i);
28 }
```

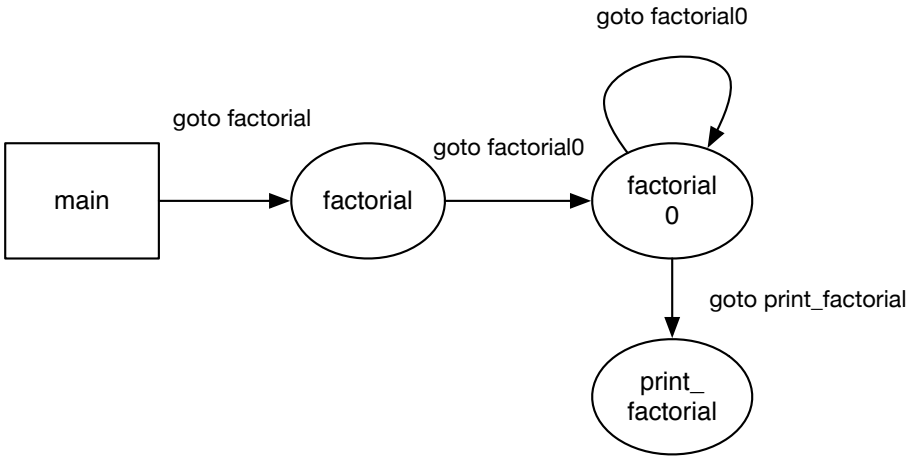

図 1.2: 階乗を求める CbC プログラム

1.3 MetaCodeSegment と MetaDataSegment

プログラムを記述する際、本来行ないたい計算の他にも記述しなければならない部分が存在する。メモリの管理やネットワーク処理、エラーハンドリングや並列処理などがこれ

にあたり、本来行ないたい計算と区別してメタ計算と呼ぶ。プログラムを動作させるためにメタ計算部分は必須であり、しばしば本来の処理よりも複雑度が高い。

CodeSegment を用いたプログラミングスタイルでは計算とメタ計算を分離して記述する。分離した計算は階層構造を持ち、本来行ないたい処理をノーマルレベルとし、メタ計算はメタレベルとしてノーマルレベルよりも上の存在に位置する。複雑なメタ計算部分をライブラリや OS 側が提供することで、ユーザはノーマルレベルの計算の記述に集中することができる。また、ノーマルレベルのプログラムに必要なメタ計算を追加することで、並列処理やネットワーク処理などを含むプログラムに拡張できる。さらに、ノーマルレベルからはメタレベルは隠蔽されているため、メタ計算の実装を切り替えることも可能である。例えば、並列処理のメタ計算用いたプログラムを作成する際、CPU で並列処理を行なうメタ計算と GPU で並列処理メタ計算を環境に応じて作成することができる。

なお、メタ計算を行なう CodeSegment は Meta CodeSegment と呼び、メタ計算に必要な DataSegment は Meta DataSegment と呼ぶ。Meta CodeSegment は CodeSegment の前後にメタ計算を挟むことで実現され、Meta DataSegment は DataSegment を含む上位の DataSegment として実現できる。よって、メタ計算は通常の計算を覆うように計算を拡張するものだと考えられる (図 1.3)。

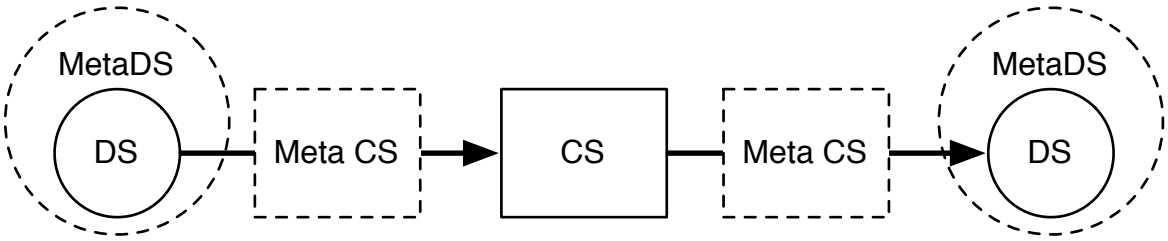

図 1.3: Meta CodeSegment と Meta DataSegment

1.4 Continuation based C におけるメタ計算の例: GearsOS

CbC を用いてメタ計算を実現した例として、GearsOS [3] が存在する。GearsOS は並列に、信頼性高く動作することを目標とした OS であり、 マルチコア CPU や GPU 環境での動作を対象としている。現在 OS の設計と並列処理部分の実装が行なわれている。GearsOS におけるメタ計算は Monad [4] を用いている。 現在実装済みのメタ計算はメモリの管理、並列に書き込むことが可能な Synchronized Queue、データの保存用の非破壊赤黒木がある。

GearsOS では CodeSegment と DataSegment はそれぞれ CodeGear と DataGear と呼ばれている。マルチコア CPU 環境では CodeGear と CodeSegment は同一だが、GPU 環境では CodeGear には OpenCL [5]/CUDA [6] における kernel も含まれる。kernel とは GPU で実行される関数のことであり、GPU 上のメモリに配置されたデータ群に対して並列に実行される。通常 GPU でデータの処理を行なう場合は

- データをメインメモリから GPU のメモリへ転送
- 転送終了を同期で確認
- kernel 起動 (GPU メモリ上のデータに対して並列に処理)
- 処理終了を同期で確認
- 計算結果であるデータを GPU のメモリからメインメモリへ転送
- 転送終了を同期で確認

といった手順が必要であり、ユーザは処理したいデータの位置などを意識しながらプログラミングする必要がある。GearsOS では CPU/GPU での処理をメタ計算としてユーザから隠すことにより、 CodeGear が実行されるデバイスや DataGear の位置を意識する必要がなくなる。

GearsOS で利用する Meta DataGear には以下のものが含まれる。

- DataGear の型情報
- DataGear を格納するメモリの情報
- CodeGear の名前と CodeGear の関数ポインタ との対応表
- CodeGear が参照する DataGear へのポインタ

実際の GearsOS におけるメモリ管理を含むメタ計算用の Meta DataGear の定義例をリスト 1.3 に示す。Meta DataGear は Context という名前の構造体で定義されている。通常レベルの DataGear も構造体で定義されているが、メタ計算側から見た DataGear はそれぞれの構造体の共用体となっており、一様に扱える。

リスト 1.3: GearsOS における Meta DataGear の定義例

```
1  /* Context definition */
2
3  #define ALLOCATE_SIZE 1024
4
5  enum Code {
6      Code1,
7      Code2,
8      Allocator,
9  };
10
11 enum UniqueData {
12     Allocate,
13     Tree,
14 };
15
16 struct Context {
17     int codeNum;
18     __code (**code) (struct Context *);
19     void* heap_start;
20     void* heap;
21     long dataSize;
22     int dataNum;
23     union Data **data;
24 };
25
26 union Data {
27     struct Tree {
28         union Data* root;
29         union Data* current;
30         union Data* prev;
31         int result;
32     } tree;
33     struct Node {
34         int key;
35         int value;
36         enum Color {
37             Red,
38             Black,
39         } color;
40         union Data* left;
41         union Data* right;
42     } node;
43     struct Allocate {
44         long size;
45         enum Code next;
46     } allocate;
```

```
47 };
```

リスト 1.3 のソースコードは以下のように対応している。

- DataGear の型情報

  DataGear は構造体を用いて定義する (リスト 1.3 27-46 行)。Tree や Node、 Allocate 構造体が DataGear に相当する。メタ計算は任意の DataGear 扱うために全ての DataGear を扱える必要がある。全ての DataGear の共用体を定義することで、DataGear を一律に扱うことができる (リスト 1.3 26-47 行)。メモリを確保する場合はこの型情報からサイズを決定する。

- DataGear を格納するメモリの情報

  メモリ領域の管理は、事前に領域を確保した後、必要に応じてその領域を割り当てることで実現する。そのために Context は割り当て済みの領域 heap と、割り当てた DataGear の数 dataNum を持つ。

- CodeGear の名前と CodeGear の関数ポインタ との対応表

  CodeGear の名前と CodeGear の関数ポインタの対応は enum と関数ポインタによって実現されている。CodeGear の名前は enum (リスト 1.3 5-9 行) で定義され、コンパイル後には整数へと変換される。プログラム全体で利用する CodeGear は code フィールドに格納されており、enum を用いてアクセスする。この対応表を動的に変更することにより、実行時に比較ルーチンなどを変更することが可能になる。

- CodeGear が参照する DataGear へのポインタ

  Meta CodeGear は Context を引数に取る CodeGear として定義されている。そのため、Meta CodeGear が DataGear の値を使う為には Context から DataGear を取り出す必要がある。取り出す必要がある DataGear は enum を用いて定義し (リスト 1.3 11-14 行)、 CodeGear を実行する際に data フィールドから取り出す。

Meta CodeGear は定義された Meta DataGear を処理する CodeGear である。メモリ管理や並列処理の待ち合わせといった処理はこのメタレベルにしか表れない。

GearsOS においては軽量継続もメタ計算として実現されている。とある CodeGear から次の CodeGear へと軽量継続する際には、次に実行される CodeGear の名前を指定する。その名前を Meta CodeGear が解釈し、対応する CodeGear へと処理を引き渡す (リスト 2.7 の `meta`)。

リスト 1.4: 通常の CodeSegment の軽量継続

```
1 __code meta(struct Context* context, enum Code next) {
2     goto (context->code[next])(context);
3 }
```

CodeGear と名前の対応は Meta DataGear に格納されており、従来の OS の Process や Thread に相当する。名前の対応を動的に切り替えたり、Thread ごとに切り替えることにより、通常レベルのプログラムを変更せず実行を上書きできる。これは従来の OS の Dynamic Loading Libary や Command の呼び出しに相当する。

また、通常レベルの CodeGear から Meta DataGear を操作できてしまうと、ユーザがメタレベル操作を自由に記述できてしまい、メタ計算を分離した意味が無くなってしまう。これを防ぐために、CodeGear を実行する際は Meta DataGear から必要な DataGear だけを渡す。このように、 Meta DataGear から DataGear を取り出す Meta CodeGear を stub と呼ぶ。stub の例をリスト 1.5 に示す。

リスト 1.5: GearsOS における stub Meta CodeSegment

```
1  __code put(struct Context* context,
2             struct Tree* tree,
3             struct Node* root,
4             struct Allocate* allocate)
5  {
6      /* ... */
7  }
8
9  __code put_stub(struct Context* context)
10 {
11     goto put(context,
12              &context->data[Tree]->tree,
13              context->data[Tree]->tree.root,
14              &context->data[Allocate]->allocate);
15 }
```

stub は Context が持つ DataGear のポインタ data に対して enum を用いてアクセスしている。なお、現在はメタレベルの計算とノーマルレベルの分離はコンパイラ側がサポートしていないため、引数に Meta DataGear である Context が渡されているが、本来はノーマルレベルではアクセスできない。

また、GearsOS におけるメタ計算として CodeGear のモデル検査がある。通常レベルの CodeGear を変更することなく、その仕様を検証するものである。個々の CodeGear の仕様を検証することにより、より信頼性の高い OS を目指す。

第2章　メタ計算ライブラリ akasha における検証

第 1 章では Continuation based C 言語の概要と、CbC で記述された GearsOS について述べた。GearsOS の持つメタ計算として、モデル検査的なアプローチで CodeGear の仕様を検証していく。

2.1　モデル検査

モデル検査とは、ソフトウェアの全ての状態において仕様が満たされるかを確認するものである。このモデル検査を行なうソフトウェアをモデル検査器と呼ぶ。モデルは検査器は、仕様の定義と確認ができる。加えて、仕様を満たさない場合にはソフトウェアがどのような状態であったか反例を返す。

モデル検査器には Spin [7] や CBMC [8] などが存在する。

Spin は Promela と呼ばれる言語でモデルを記述し、その中に論理式として仕様を記述する。論理式は assert でモデルの内部に埋め込まれ、並列に実行してもその仕様が満たされるかをチェックする。また、Promela で記述されたモデルから C 言語を生成することができる。しかし、Promela で記述されたモデルは元の C 言語とはかなり異なる構文をしており、ユーザが記述する難易度が高い。

そこで、モデルを個別に記述せずに実装そのものを検査するアプローチがある。例えばモデル検査器 CBMC は C 言語を直接検証できる。CBMC でも仕様は論理式で記述され、 assert と組み合わせる。C 言語の実行は通常の実行とは異なり、記号実行という形で実行される。プログラム上の変数は記号として処理され、`a < b` といった条件式により分岐が行なわれたのなら、その条件を持つ場合の経路、持たない場合の経路、と分岐していくのである。

GearsOS におけるモデル検査的なアプローチは CBMC のように実装言語をそのまま検証できるようにしたい。そのために、assert を利用した仕様の定義と、その検査、必要なら反例を提出するようなメタ計算を定義する。このメタ計算をメタ計算ライブラリ akasha として実装した。

この章では、メタ計算ライブラリ akasha を用いて GearsOS のデータ構造を検証していく。

2.2　GearsOS における非破壊赤黒木

現状の GearsOS に実装されているメタ計算として、非破壊赤黒木が存在する。非破壊赤黒木はユーザがデータを保存する際に利用することを想定している。メタ計算として定義することで、ノーマルレベルからは木のバランスを考慮せず木への操作が行なえる。

なお、赤黒木とは二分探索木の一種であり、木のバランスを取るための情報として各ノードは赤か黒の色を持っている。

二分探索木の条件は以下である。

- 左の子孫の値は親の値より小さい
- 右の子孫の値は親の値より大きい

加えて、赤黒木が持つ具体的な条件は以下のものである。

- 各ノードは赤か黒の色を持つ。
- ルートノードの色は黒である。
- 葉ノードの色は黒である。
- 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無い)。
- 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。

数値を要素に持つ赤黒木の例を図 2.1 に示す。条件に示されている通り、ルートノードは黒であり、赤ノードは連続していない。加えて各最下位ノードへの経路に含まれる黒ノードの個数は全て 2 である。

赤黒木の持つ条件を言い変えるのなら、「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二倍に収まる」とも言える。この言い換えは「赤が続くことはない」という条件と「ルートから最下位への経路の黒ノードはどの最下位ノードでも同じ」であることから導ける。具体的には、最短経路は「黒のみの経路」であり、最長経路は「黒と赤が交互に続く経路」となる。この条件を言い変えた性質を仕様とし、検証していく。

GearsOS で実装されている赤黒木は特に非破壊赤黒木であり、一度構築した木構造は破壊される操作ごとに新しい木構造が生成される。非破壊の性質を付与した理由として、

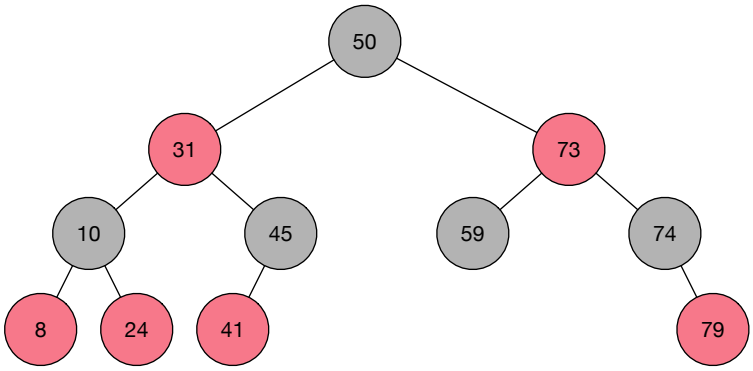

図 2.1: 赤黒木の例

並列実行時のデータの保存がある。同じ赤黒木をロックせずに同時に更新した場合、ノードの値は実行順に依存したり、競合したりする。しかし、ロックを行なって更新した場合は同じ木に対する処理に待ち合わせが発生し、全体の並列度が下がる。この問題に対し GearsOS では、各スレッドは処理を行なう際には非破壊の木を利用することで並列度は保ち、値の更新が発生する時のみ木をアトミックな操作で置き換えることで競合を回避する。具体的には木の操作を行なった後はルートのノードを元に CAS で置き換え、失敗した時は木を読み込み直して処理を再実行する。CAS(Check and Set) とは、アトミックに値を置き換える操作であり、使う際は更新前の値と更新後の値を渡す。CAS で渡された更新前の値が、保存している値と同じであれば競合していないために値の更新に成功し、異なる場合は他に書き込みがあったことを示すために値の更新が失敗する操作のことである。

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、変更後に新たなルートノードとする。この際に変更が行なわれていない部分は変更前の木と共有する (図 2.2)。これは一度構築された木構造は破壊されないという非破壊の性質を用いたメモリ使用量の最適化である。

CbC を用いて赤黒木を実装する際の問題として、関数の呼び出しスタックが存在しないことがある。C における実装では関数の再帰呼び出しによって木が辿るが、それが行なえない。経路を辿るためには、ノードに親への参照を持たせるか、挿入や削除の際に辿った経路を記憶する必要がある。ノードが親への参照を持つ非破壊木構造は共通部分の共有が行なえないため、経路を記憶する方法を使う。経路の記憶にはスタックを用い、スタックは Meta DataSegment に保持する。

赤黒木を格納する DataSegment と Meta DataSegment の定義をリスト 2.1 に示す。経路の記憶に用いるスタックは Meta DataSegment である Context 内部の `node_stack` である。DataSegment は各ノード情報を持つ `Node` 構造体と、赤黒木を格納する `Tree` 構造

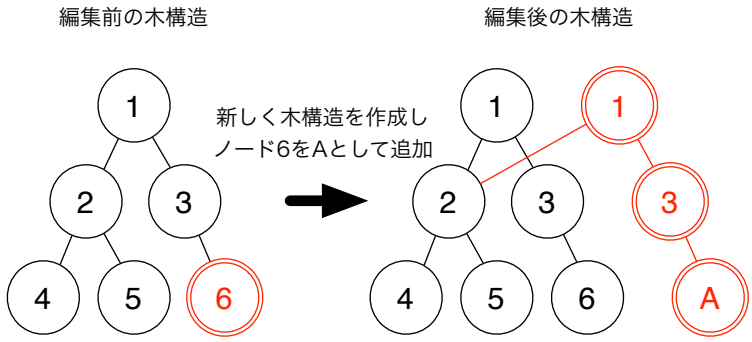

図 2.2: 非破壊赤黒木の編集

体、挿入などで操作中の一時的な木を格納する `Traverse` 共用体などがある。

リスト 2.1: 赤黒木の DataSegment と Meta DataSegment

```
1  // DataSegments for Red-Black Tree
2  union Data {
3      struct Comparable { // interface
4          enum Code compare;
5          union Data* data;
6      } compare;
7      struct Count {
8          enum Code next;
9          long i;
10     } count;
11     struct Tree {
12         enum Code next;
13         struct Node* root;
14         struct Node* current;
15         struct Node* deleted;
16         int result;
17     } tree;
18     struct Node {
19         // need to tree
20         enum Code next;
21         int key; // comparable data segment
22         int value;
23         struct Node* left;
24         struct Node* right;
25         // need to balancing
26         enum Color {
27             Red,
28             Black,
29         } color;
30     } node;
31     struct Allocate {
32         enum Code next;
33         long size;
34     } allocate;
```

```
35 };
36
37
38 // Meta DataSegment
39 struct Context {
40     enum Code next;
41     int codeNum;
42     __code (**code) (struct Context*);
43     void* heapStart;
44     void* heap;
45     long heapLimit;
46     int dataNum;
47     stack_ptr code_stack;
48     stack_ptr node_stack;
49     union Data **data;
50 };
```

Meta DataSegment を初期化する Meta CodeSegment initLLRBContext をリスト 2.2 に示す。この Meta CodeSegment ではメモリ領域の確保、CodeSegment 名と CodeSegment の実体の対応表の作成などを行なう。メモリ領域はプログラムの起動時に一定数のメモリを確保し、ヒープとして `heap` フィールドに保持させる。CodeSegment 名と CodeSegment の実体との対応は、enum で定義された CodeSegment 名の添字へと CodeSegment の関数ポインタを代入することにより持つ。例えば `Put` の実体は `put_stub` である。他にも DataSegment の初期化 (リスト 2.2 34-48) とスタックの初期化 (リスト 2.2 50-51) を行なう。

リスト 2.2: 赤黒木の Meta DataSegment の初期化を行なう Meta CodeSegment

```
1  __code initLLRBContext(struct Context* context, int num) {
2      context->heapLimit = sizeof(union Data)*ALLOCATE_SIZE;
3      context->code = malloc(sizeof(__code*)*ALLOCATE_SIZE);
4      context->data = malloc(sizeof(union Data*)*ALLOCATE_SIZE);
5      context->heapStart = malloc(context->heapLimit);
6
7      context->codeNum = Exit;
8
9      context->code[Code1]      = code1_stub;
10     context->code[Code2]      = code2_stub;
11     context->code[Code3]      = code3_stub;
12     context->code[Code4]      = code4;
13     context->code[Code5]      = code5;
14     context->code[Find]       = find;
15     context->code[Not_find]   = not_find;
16     context->code[Code6]      = code6;
17     context->code[Put]        = put_stub;
18     context->code[Replace]    = replaceNode_stub;
19     context->code[Insert]     = insertNode_stub;
20     context->code[RotateL]    = rotateLeft_stub;
21     context->code[RotateR]    = rotateRight_stub;
22     context->code[InsertCase1]   = insert1_stub;
23     context->code[InsertCase2]   = insert2_stub;
24     context->code[InsertCase3]   = insert3_stub;
```

```
25      context->code[InsertCase4]   = insert4_stub;
26      context->code[InsertCase4_1] = insert4_1_stub;
27      context->code[InsertCase4_2] = insert4_2_stub;
28      context->code[InsertCase5]   = insert5_stub;
29      context->code[StackClear]    = stackClear_stub;
30      context->code[Exit]       = exit_code;
31
32      context->heap = context->heapStart;
33
34      context->data[Allocate] = context->heap;
35      context->heap += sizeof(struct Allocate);
36
37      context->data[Tree] = context->heap;
38      context->heap += sizeof(struct Tree);
39
40      context->data[Node] = context->heap;
41      context->heap += sizeof(struct Node);
42
43      context->dataNum = Node;
44
45      struct Tree* tree = &context->data[Tree]->tree;
46      tree->root = 0;
47      tree->current = 0;
48      tree->deleted = 0;
49
50      context->node_stack = stack_init(sizeof(struct Node*), 100);
51      context->code_stack = stack_init(sizeof(enum Code), 100);
52  }
```

実際の赤黒木の実装に用いられている Meta CodeSegment の一例をリスト 2.3 に示す。Meta CodeSegment `insertCase2` は要素を挿入した場合に呼ばれる Meta CodeSegment の一つであり、親ノードの色によって処理を変える。まず、色を確認するために経路を記憶しているスタックから親の情報を取り出す。親の色が黒ならば処理を終了し、次の CodeSegment へと軽量継続する (リスト 2.3 5-8)。親の色が赤であるならばさらに処理を続行して `InsertCase3` へと軽量継続する。ここで、経路情報を再現するためにスタックへと親を再代入してから軽量継続を行なっている。なお、Meta CodeSegment でも Context から DataSegment を展開する処理は stub によって行なわれる (リスト 2.3 14-16)。

リスト 2.3: 赤黒木の実装に用いられている Meta CodeSegment 例

```
1   __code insertCase2(struct Context* context, struct Node* current) {
2       struct Node* parent;
3       stack_pop(context->node_stack, &parent);
4
5       if (parent->color == Black) {
6           stack_pop(context->code_stack, &context->next);
7           goto meta(context, context->next);
8       }
9
10      stack_push(context->node_stack, &parent);
11      goto meta(context, InsertCase3);
```

```
12 }
13
14 __code insert2_stub(struct Context* context) {
15     goto insertCase2(context, context->data[Tree]->tree.current);
16 }
```

2.3　メタ計算ライブラリ akasha を用いた赤黒木の実装の検証

赤黒木の仕様の定義とその確認を CbC で行なっていく。仕様には赤黒木の利用方法などによっていくつかのものが考えられる。赤黒木に対する操作の仕様と、その操作によって保証されるべき赤黒木の状態を示すと以下のようになる。

- 挿入したデータは参照できること
- 削除したデータは参照できないこと
- 値を更新した後は更新された値が参照されること
- 操作を行なった後の木はバランスしていること

今回はバランスに関する仕様を確認する。操作を挿入に限定し、どのような順番で要素を挿入しても木がバランスすることを検証する。検証には当研究室で開発しているメタ計算ライブラリ akasha を用いる。

akasha では仕様は常に成り立つべき CbC の条件式として定義される。具体的には Meta CodeSegment に定義した assert が仕様に相当する。仕様の例として「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」という式を定義する (リスト 2.4)。

リスト 2.4: 木の高さに関する仕様記述

```
1 void verifySpecification(struct Context* context, struct Tree* tree) {
2     assert(!(maxHeight(tree->root, 1) > 2*minHeight(tree->root, 1)));
3     return meta(context, EnumerateInputs);
4 }
```

リスト 2.4 で定義した仕様がプログラムの持つ全ての状態に成り立つかを確認する。また、成り立たない場合には仕様に反する状態を反例として提出する。

まずは最も単純な検証として要素数を有限に固定し、その挿入順番を数え上げる。最初に、検証の対象となる赤黒木と、検証に必要な DataSegment を含む Meta DataSegment を定義する (リスト 2.5)。これが akasha のレベルで利用する Meta DataSegment である。赤黒木自体はユーザから見るとメタレベル計算であるが、今回はその実装の検証するた

め、赤黒木がノーマルレベルとなる。よって akasha はメタメタレベルの計算とも考えられる。

akasha が使う DataSegment は データの挿入順を数え上げるためには使う環状リスト `Iterator` とその要素 `IterElem`、検証に使う情報を保持する `AkashaInfo`、木をなぞる際に使う `AkashaNode` がある。

リスト 2.5: 検証を行なうための Meta DataSegment

```
1  // Data Segment
2  union Data {
3      struct Tree { /* ... */ } tree;
4      struct Node { /* ... */ } node;
5
6      /* for verification */
7      struct IterElem {
8          unsigned int val;
9          struct IterElem* next;
10     } iterElem;
11     struct Iterator {
12         struct Tree* tree;
13         struct Iterator* previousDepth;
14         struct IterElem* head;
15         struct IterElem* last;
16         unsigned int  iteratedValue;
17         unsigned long iteratedPointDataNum;
18         void*         iteratedPointHeap;
19     } iterator;
20     struct AkashaInfo {
21         unsigned int minHeight;
22         unsigned int maxHeight;
23         struct AkashaNode* akashaNode;
24     } akashaInfo;
25     struct AkashaNode {
26         unsigned int       height;
27         struct Node*       node;
28         struct AkashaNode* nextAkashaNode;
29     } akashaNode;
30 };
```

挿入順番の数え上げには環状リストを用いた深さ優先探索を用いる。最初に検証する要素を全て持つ環状リストを作成し、木に挿入した要素を除きながら環状リストを複製していく。環状リストが空になった時が組み合わせを一つ列挙し終えた状態となる。列挙し終えた後、前の深さの環状リストを再現してリストの先頭を進めることで異なる組み合わせを列挙する。

仕様には木の高さが含まれるので、高さを取得する Meta CodeSegment が必要となる。リスト 2.6 に木の最も低い経路の長さを取得する Meta CodeSegment を示す。

木を辿るためのスタックに相当する `AkshaNode` を用いて経路を保持しつつ、高さを確認している。スタックが空であれば全てのノードを確認したので次の CodeSegment へと

軽量継続を行なう。空でなければ今辿っているノードが葉であるか確認し、葉ならば高さを更新して次のノードを確認するため自身へと軽量継続する。葉でなければ高さを 1 増やして左右の子をスタックに積み、自身へと軽量継続を行なう。

リスト 2.6: 木の最も短かい経路の長さを確認する Meta CodeSegment

```
1  __code getMinHeight_stub(struct Context* context) {
2      goto getMinHeight(context, &context->data[Allocate]->allocate, &
       context->data[AkashaInfo]->akashaInfo);
3  }
4
5  __code getMinHeight(struct Context* context, struct Allocate* allocate,
       struct AkashaInfo* akashaInfo) {
6      const struct AkashaNode* akashaNode = akashaInfo->akashaNode;
7
8      if (akashaNode == NULL) {
9          allocate->size = sizeof(struct AkashaNode);
10         allocator(context);
11         akashaInfo->akashaNode = (struct AkashaNode*)context->data[
       context->dataNum];
12
13         akashaInfo->akashaNode->height = 1;
14         akashaInfo->akashaNode->node   = context->data[Tree]->tree.root;
15
16         goto getMaxHeight_stub(context);
17     }
18
19     const struct Node* node = akashaInfo->akashaNode->node;
20     if (node->left == NULL && node->right == NULL) {
21         if (akashaInfo->minHeight > akashaNode->height) {
22             akashaInfo->minHeight  = akashaNode->height;
23             akashaInfo->akashaNode = akashaNode->nextAkashaNode;
24             goto getMinHeight_stub(context);
25         }
26     }
27
28     akashaInfo->akashaNode = akashaInfo->akashaNode->nextAkashaNode;
29
30     if (node->left != NULL) {
31         allocate->size = sizeof(struct AkashaNode);
32         allocator(context);
33         struct AkashaNode* left = (struct AkashaNode*)context->data[
       context->dataNum];
34         left->height           = akashaNode->height+1;
35         left->node             = node->left;
36         left->nextAkashaNode   = akashaInfo->akashaNode;
37         akashaInfo->akashaNode = left;
38     }
39
40     if (node->right != NULL) {
41         allocate->size = sizeof(struct AkashaNode);
42         allocator(context);
43         struct AkashaNode* right = (struct AkashaNode*)context->data[
```

```
    context->dataNum];
44          right->height             = akashaNode->height+1;
45          right->node               = node->right;
46          right->nextAkashaNode     = akashaInfo->akashaNode;
47          akashaInfo->akashaNode    = right;
48      }
49
50      goto getMinHeight_stub(context);
51  }
```

同様に最も高い高さを取得し、仕様であるリスト 2.4 の assert を挿入の度に実行する。assert は CodeSegment の結合を行なうメタ計算である `meta` を上書きすることにより実現する。

`meta` はリスト 2.3 の `insertCase2` のように軽量継続を行なう際に CodeSegment 名と DataSegment を指定するものである。検証を行なわない通常の `meta` の実装は CodeSegment 名から対応する実体への軽量継続である (リスト 2.7)。

リスト 2.7: 通常の CodeSegment の軽量継続

```
1  __code meta(struct Context* context, enum Code next) {
2      goto (context->code[next])(context);
3  }
```

これを、検証を行なうように変更することで `insertCase2` といった赤黒木の実装のコードを修正することなく検証を行なうことができる。検証を行ないながら軽量継続する `meta` はリスト 2.8 のように定義される。実際の検証部分は `PutAndGoToNextDepth` の後に行なわれるため、直接は記述されていない。この `meta` が行なうのは検証用にメモリの管理である。状態の数え上げを行なう際に状態を保存したり、元の状態に戻す処理が行なわれる。このメタ計算を用いた検証では、要素数 13 個までの任意の順で挿入の際に仕様が満たされることを確認できた。また、赤黒木の処理内部に恣意的なバグを追加した際には反例を返した。

リスト 2.8: 検証を行なう CodeSegment の軽量継続

```
1  __code meta(struct Context* context, enum Code next) {
2      struct Iterator* iter = &context->data[Iter]->iterator;
3
4      switch (context->prev) {
5          case GoToPreviousDepth:
6              if (iter->iteratedPointDataNum == 0) break;
7              if (iter->iteratedPointHeap == NULL) break;
8
9              unsigned int diff =(unsigned long)context->heap - (unsigned
     long)iter->iteratedPointHeap;
10             memset(iter->iteratedPointHeap, 0, diff);
11             context->dataNum = iter->iteratedPointDataNum;
12             context->heap    = iter->iteratedPointHeap;
13             break;
14         default:
```

```
15              break;
16      }
17      switch (next) {
18          case PutAndGoToNextDepth:   // with assert check
19              if (context->prev == GoToPreviousDepth) break;
20              if (iter->previousDepth == NULL)        break;
21              iter->previousDepth->iteratedPointDataNum = context->dataNum;
22              iter->previousDepth->iteratedPointHeap    = context->heap;
23              break;
24          default:
25              break;
26      }
27
28      context->prev = next;
29      goto (context->code[next])(context);
30  }
```

2.4　モデル検査器 CBMC との比較

akasha の比較対象として、C 言語の有限モデルチェッカ CBMC [?] を用いて赤黒木を検証した。CBMC は ANSI-C を記号実行し、仕様の否定となるような実行パターンが無いかを検証するツールである。

比較のために全く同じ赤黒木のソースコードを用いたいが、CbC の構文は厳密には C とは異なるために変換が必要である。具体的には、`__code` を `void` に、`goto` を `return` に置換することで機械的に C 言語に変換できる。

CBMC における仕様は bool を返す式として記述するため、akasha と同様の仕様定義が利用できる (リスト 2.9。assert が true になるような実行パターンを CBMC が見付けると、その実行パターンが反例として出力される。

リスト 2.9: CBMC における仕様記述

```
1  void verifySpecification(struct Context* context,
2                           struct Tree* tree) {
3      assert(!(maxHeight(tree->root, 1) >
4               2*minHeight(tree->root, 1)));
5      return meta(context, EnumerateInputs);
6  }
```

挿入順の数え上げには CBMC の機能に存在する非決定的な値 `nondet_int()` を用いた (リスト 2.10)。この `nondet_int()` 関数は int の持ちうる値の内から非決定的に値を取得する関数である。akasha では有限の要素個分の組み合わせを用いて挿入順の数え上げとしたが、CBMC では要素数回分だけランダムな値を入力させることで数え上げとする。

リスト 2.10: CBMC における挿入順の数え上げ

```
1  void enumerateInputs(struct Context* context,
2                       struct Node* node) {
3      if (context->loopCount > LIMIT_OF_VERIFICATION_SIZE) {
4          return meta(context, Exit);
5      }
6
7      node->key     = nondet_int();
8      node->value   = node->key;
9      context->next = VerifySpecification;
10     context->loopCount++;
11
12     return meta(context, Put);
13 }
```

CBMC では有限のステップ数だけ C 言語を記号実行し、その範囲内で仕様が満たされるかを確認する。条件分岐や繰り返しなどは展開されて実行される。基本的にはメモリの許す限り展開を行なうことができるが、今回の赤黒木の検証では 411 回まで展開することができた。この 411 回のうちの実行パスでは赤黒木の仕様は常に満たされる。しかし、この展開された回数は挿入された回数とは無関係であり、実際どの程度検証することができたか確認できない。実際、赤黒木に恣意的なバグを追加した際にも仕様の反例は得られず、CBMC で扱える範囲内では赤黒木の性質は検証できなかった。

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

謝辞

本研究の遂行、本論文の作成にあたり、御多忙にも関わらず終始懇切なる御指導と御教授を賜わりました河野真治准教授に心より感謝致します。そして、 共に研究を行い暖かな気遣いと励ましをもって支えてくれた並列信頼研究室の全てのメンバーに感謝致します。最後に、 有意義な時間を共に過ごした理工学研究科情報工学専攻の学友、並びに物心両面で支えてくれた家族に深く感謝致します。

2017 年 3 月
比嘉健太

参考文献

[1] Tokumori Kaito and Kono Shinji. The implementation of continuation based c compiler on llvm/clang 3.5. *IPSJ SIG Notes*, Vol. 2014, No. 10, pp. 1–11, may 2014.

[2] 信康大城, 真治河野. Continuation based c の gcc4.6 上の実装について. 第 53 回プログラミング・シンポジウム予稿集, 第 2012 巻, pp. 69–78, jan 2012.

[3] 翔平小久保, 立樹伊波, 真治河野. Monad に基づくメタ計算を基本とする gears os の設計. Technical Report 16, 琉球大学大学院理工学研究科情報工学専攻, 琉球大学工学部情報工学科, 琉球大学工学部情報工学科, may 2015.

[4] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, Vol. 93, No. 1, pp. 55–92, July 1991.

[5] Opencl — nvidia developer. `https://developer.nvidia.com/opencl`. Accessed: 2016/02/06(Mon).

[6] Cuda zone — nvidia developer. `https://developer.nvidia.com/cuda-zone`. Accessed: 2016/02/06(Mon).

[7] Spin - formal verification. `http://spinroot.com/spin/whatispin.html`. Accessed: 2016/01/20(Fri).

[8] The cbmc homepage. `http://www.cprover.org/cbmc/`. Accessed: 2016/01/20(Fri).

[9] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[10] Joachim (mathématicien) Lambek and P. J. Scott. *Introduction to higher order categorical logic*. Cambridge studies in advanced mathematics. Cambridge University Press, Cambridge, New York (N. Y.), Melbourne, 1986.

[11] Michael Barr and Charles Wells. *Category Theory for Computing Science*. International Series in Computer Science. Prentice-Hall, 1990. Second edition, 1995.

[12] M. P. Jones and L. Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, December 1993.

[13] The agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`. Accessed: 2016/01/20(Fri).

[14] Welcome to agda's documentation! — agda 2.6.0 documentation. `http://agda.readthedocs.io/en/latest/index.html`. Accessed: 2016/01/31(Tue).

[15] Welcome! — the coq proof assistant. `https://coq.inria.fr/`. Accessed: 2016/01/20(Fri).

[16] Ats-pl-sys. `http://www.ats-lang.org/`. Accessed: 2016/01/20(Fri).

[17] Nusmv home page. `http://nusmv.fbk.eu/`. Accessed: 2016/01/20(Fri).

[18] 徳森海斗. Llvm clang 上の continuation based c コンパイラ の改良. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2016.

[19] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[20] B.C. Pierce. 型システム入門プログラミング言語と型の理論:. オーム社, 2013.

[21] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pp. 1–2, New York, NY, USA, 2009. ACM.

[22] John Backus. The history of fortran i, ii, and iii. *SIGPLAN Not.*, Vol. 13, No. 8, pp. 165–180, August 1978.

[23] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, Vol. 6, No. 4, pp. 308–320, January 1964.

[24] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jork, 1941.

[25] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, Vol. 27, No. 5, May 1992.

[26] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, Vol. 75, No. 5, pp. 381 – 392, 1972.

発表履歴

- 比嘉健太, 河野真治. 形式手法を学び始めて思うことと、形式手法を広めるには. 情報処理学会ソフトウェア工学研究会 (IPSJ SIGSE) ウィンターワークショップ 2015・イン・宜野湾 (WWS2015), Jan 2015.
- 比嘉健太, 河野真治. Continuation based C を用いたプログラムの検証手法. 2016 年並列／分散／協調処理に関する『松本』サマー・ワークショップ (SWoPP2016) 情報処理学会・プログラミング研究会 第 110 回プログラミング研究会 (PRO-2016-2) Aug 2016.

付 録A　ソースコード一覧

本論文中に取り上げた Agda の動作するソースコードを示す。

A-1　部分型の定義

リスト A.1 に Agda 上で定義した CbC の部分型の定義を示す。

リスト A.1: Agda 上で定義した CbC の部分型の定義 (subtype.agda)

```
1  open import Level
2  open import Relation.Binary.PropositionalEquality
3
4  module subtype {l : Level} (Context : Set l) where
5
6
7  record DataSegment {ll : Level} (A : Set ll) : Set (l ^^e2^^8a^^94 ll)
       where
8    field
9      get : Context -> A
10     set : Context -> A -> Context
11 open DataSegment
12
13 data CodeSegment {l1 l2 : Level} (A : Set l1) (B : Set l2) : Set (l ^^e2
       ^^8a^^94 l1 ^^e2^^8a^^94 l2) where
14   cs : {{_ : DataSegment A}} {{_ : DataSegment B}} -> (A -> B) ->
       CodeSegment A B
15
16 goto : {l1 l2 : Level} {I : Set l1} {O : Set l2} -> CodeSegment I O -> I
       -> O
17 goto (cs b) i = b i
18
19 exec : {l1 l2 : Level} {I : Set l1} {O : Set l2} {{_ : DataSegment I}} {{
       _ : DataSegment O}}
20      -> CodeSegment I O -> Context -> Context
21 exec {l} {{i}} {{o}}  (cs b) c = set o c (b (get i c))
22
23
24 comp : {con : Context} -> {l1 l2 l3 l4 : Level}
25        {A : Set l1} {B : Set l2} {C : Set l3} {D : Set l4}
26        {{_ : DataSegment A}} {{_ : DataSegment B}} {{_ : DataSegment C}}
       {{_ : DataSegment D}}
27        -> (C -> D) -> (A -> B) -> A -> D
```

```
28 comp {con} {{i}} {{io}} {{oi}} {{o}} g f x = g (get oi (set io con (f x))
       )
29
30 csComp : {con : Context} -> {l1 l2 l3 l4 : Level}
31         {A : Set l1} {B : Set l2} {C : Set l3} {D : Set l4}
32          {{_ : DataSegment A}} {{_ : DataSegment B}} {{_ : DataSegment C
       }} {{_ : DataSegment D}}
33         -> CodeSegment C D -> CodeSegment A B -> CodeSegment A D
34 csComp {con} {A} {B} {C} {D} {{da}} {{db}} {{dc}} {{dd}} (cs g) (cs f)
35       = cs {{da}} {{dd}} (comp {con} {{da}} {{db}} {{dc}} {{dd}} g f)
36
37
38
39 comp-associative : {A B C D E F : Set l} {con : Context}
40                    {{da : DataSegment A}} {{db : DataSegment B}} {{dc :
       DataSegment C}}
41                    {{dd : DataSegment D}} {{de : DataSegment E}} {{df :
       DataSegment F}}
42                    -> (a : CodeSegment A B) (b : CodeSegment C D) (c :
       CodeSegment E F)
43                    -> csComp {con} c (csComp {con} b a)  ≡
        csComp {con} (csComp {con} c b) a
44 comp-associative (cs _) (cs _) (cs _) = refl
```

A-2　ノーマルレベル計算の実行

??節で取り上げたソースコードをリスト A.2 に示す。CbC のコードにより近づけるように A gda 上の `Data.Nat` を `Int` という名前に変更している。

リスト A.2: ノーマルレベル計算例の完全なソースコード (atton-master-sample.agda)

```
1  module atton-master-sample where
2
3  open import Data.Nat
4  open import Data.Unit
5  open import Function
6  Int = ℕ
7
8  record Context : Set where
9    field
10     a : Int
11     b : Int
12     c : Int
13
14
15 open import subtype Context
16
17
18
19 record ds0 : Set where
20   field
```

```
21      a : Int
22      b : Int
23
24  record ds1 : Set where
25    field
26      c : Int
27
28  instance
29    _ : DataSegment ds0
30    _ = record { set = (\c d → record c {a = (ds0.a d) ; b = (ds0.b d)})
31               ; get = (\c →   record { a = (Context.a c) ; b = (Context.b
      c)})}
32    _ : DataSegment ds1
33    _ = record { set = (\c d → record c {c = (ds1.c d)})
34               ; get = (\c →   record { c = (Context.c c)})}
35
36  cs2 : CodeSegment ds1 ds1
37  cs2 = cs id
38
39  cs1 : CodeSegment ds1 ds1
40  cs1 = cs (\d → goto cs2 d)
41
42  cs0 : CodeSegment ds0 ds1
43  cs0 = cs (\d → goto cs1 (record {c = (ds0.a d) + (ds0.b d)}))
44
45  main : ds1
46  main = goto cs0 (record {a = 100 ; b = 50})
```

A-3　メタレベル計算の実行

??節で取り上げたソースコードをリスト A.3 に示す。

リスト A.3: メタレベル計算例の完全なソースコード (atton-master-meta-sample.agda)

```
1   module atton-master-meta-sample where
2
3   open import Data.Nat
4   open import Data.Unit
5   open import Function
6   Int = ℕ
7
8   record Context : Set where
9     field
10      a  : Int
11      b  : Int
12      c  : Int
13
14  open import subtype Context as N
15
16  record Meta : Set where
17    field
18      context : Context
```

```
19      c'      : Int
20      next    : N.CodeSegment Context Context
21
22  open import subtype Meta as M
23
24  instance
25    _ : N.DataSegment Context
26    _ = record { get = id ; set = (\_ c → c) }
27    _ : M.DataSegment Context
28    _ = record { get = (\m → Meta.context m)  ;
29                 set = (\m c → record m {context = c}) }
30    _ : M.DataSegment Meta
31    _ = record { get = id ; set = (\_ m → m) }
32
33
34  liftContext : {X Y : Set} {{_ : N.DataSegment X}} {{_ : N.DataSegment Y}}
        → N.CodeSegment X Y → N.CodeSegment Context Context
35  liftContext {{x}} {{y}} (N.cs f) = N.cs (\c → N.DataSegment.set y c (f (
        N.DataSegment.get x c)))
36
37  liftMeta : {X Y : Set} {{_ : M.DataSegment X}} {{_ : M.DataSegment Y}}
        → N.CodeSegment X Y → M.CodeSegment X Y
38  liftMeta (N.cs f) = M.cs f
39
40
41  gotoMeta : {I O : Set} {{_ : N.DataSegment I}} {{_ : N.DataSegment O}}
        → M.CodeSegment Meta Meta → N.CodeSegment I O → Meta → Meta
42  gotoMeta mCode code m = M.exec mCode (record m {next = (liftContext code)
        })
43
44  push : M.CodeSegment Meta Meta
45  push = M.cs (\m → M.exec (liftMeta (Meta.next m)) (record m {c' =
        Context.c (Meta.context m)}))
46
47
48  record ds0 : Set where
49    field
50      a : Int
51      b : Int
52
53  record ds1 : Set where
54    field
55      c : Int
56
57  instance
58    _ : N.DataSegment ds0
59    _ = record { set = (\c d → record c {a = (ds0.a d) ; b = (ds0.b d)})
60                ; get = (\c →   record { a = (Context.a c) ; b = (Context.b
          c)})}
61    _ : N.DataSegment ds1
62    _ = record { set = (\c d → record c {c = (ds1.c d)})
63                ; get = (\c →   record { c = (Context.c c)})}
64
65  cs2 : N.CodeSegment ds1 ds1
66  cs2 = N.cs id
```

```
67
68 cs1 : N.CodeSegment ds1 ds1
69 cs1 = N.cs (\d → N.goto cs2 d)
70
71 cs0 : N.CodeSegment ds0 ds1
72 cs0 = N.cs (\d → N.goto cs1 (record {c = (ds0.a d) + (ds0.b d)}))
73
74
75 main : Meta
76 main = gotoMeta push cs0 (record {context = (record {a = 100 ; b = 50 ; c
       = 70}) ; c' = 0 ; next = (N.cs id)})
77 -- record {context = record {a = 100 ; b = 50 ; c = 150} ; c' = 70 ; next
       = (N.cs id)}
```

A-4　Agda を用いた Continuation based C の検証

??節で取り上げたソースコードを以下に示す。

リスト A.4: Agda を用いた Continuation based C の検証コード (SingleLinkedStack.cbc)

```
1  #include "../context.h"
2  #include "../origin_cs.h"
3  #include <stdio.h>
4
5  // typedef struct SingleLinkedStack {
6  //     struct Element* top;
7  // } SingleLinkedStack;
8
9  Stack* createSingleLinkedStack(struct Context* context) {
10     struct Stack* stack = new Stack();
11     struct SingleLinkedStack* singleLinkedStack = new SingleLinkedStack()
       ;
12     stack->stack = (union Data*)singleLinkedStack;
13     singleLinkedStack->top = NULL;
14     stack->push = C_pushSingleLinkedStack;
15     stack->pop  = C_popSingleLinkedStack;
16     stack->pop2  = C_pop2SingleLinkedStack;
17     stack->get  = C_getSingleLinkedStack;
18     stack->get2  = C_get2SingleLinkedStack;
19     stack->isEmpty = C_isEmptySingleLinkedStack;
20     stack->clear = C_clearSingleLinkedStack;
21     return stack;
22 }
23
24 void printStack1(union Data* data) {
25     struct Node* node = &data->Element.data->Node;
26     if (node == NULL) {
27         printf("NULL");
28     } else {
29         printf("key = %d ,", node->key);
30         printStack1((union Data*)data->Element.next);
```

```
31      }
32  }
33
34  void printStack(union Data* data) {
35      printStack1(data);
36      printf("\n");
37  }
38
39  __code clearSingleLinkedStack(struct SingleLinkedStack* stack,__code next
        (...)) {
40      stack->top = NULL;
41      goto next(...);
42  }
43
44  // TODO
45  __code pushSingleLinkedStack(struct SingleLinkedStack* stack,union Data*
        data, __code next(...)) {
46      Element* element = new Element();
47      element->next = stack->top;
48      element->data = data;
49      stack->top = element;
50      goto next(...);
51  }
52
53  __code popSingleLinkedStack(struct SingleLinkedStack* stack, __code next(
        union Data* data, ...)) {
54      if (stack->top) {
55          data = stack->top->data;
56          stack->top = stack->top->next;
57      } else {
58          data = NULL;
59      }
60      goto next(data, ...);
61  }
62
63  __code pop2SingleLinkedStack(struct SingleLinkedStack* stack, __code next
        (union Data* data, union Data* data1, ...)) {
64      if (stack->top) {
65          data = stack->top->data;
66          stack->top = stack->top->next;
67      } else {
68          data = NULL;
69      }
70      if (stack->top) {
71          data1 = stack->top->data;
72          stack->top = stack->top->next;
73      } else {
74          data1 = NULL;
75      }
76      goto next(data, data1, ...);
77  }
78
79
```

```
80 __code getSingleLinkedStack(struct SingleLinkedStack* stack, __code next(
       union Data* data, ...)) {
81     if (stack->top)
82         data = stack->top->data;
83     else
84         data = NULL;
85     goto next(data, ...);
86 }
87
88 __code get2SingleLinkedStack(struct SingleLinkedStack* stack, __code next
       (union Data* data, union Data* data1, ...)) {
89     if (stack->top) {
90         data = stack->top->data;
91         if (stack->top->next) {
92             data1 = stack->top->next->data;
93         } else {
94             data1 = NULL;
95         }
96     } else {
97         data = NULL;
98         data1 = NULL;
99     }
100    goto next(data, data1, ...);
101 }
102
103 __code isEmptySingleLinkedStack(struct SingleLinkedStack* stack, __code
       next(...), __code whenEmpty(...)) {
104    if (stack->top)
105        goto next(...);
106    else
107        goto whenEmpty(...);
108 }
```

リスト A.5: Agda を用いた Continuation based C の検証コード (stack-subtype.agda)

```
1  open import Level hiding (lift)
2  open import Data.Maybe
3  open import Data.Product
4  open import Data.Nat hiding (suc)
5  open import Function
6
7  module stack-subtype (A : Set) where
8
9  -- data definitions
10
11 data Element (a : Set) : Set where
12   cons : a → Maybe (Element a) → Element a
13
14 datum : {a : Set} → Element a → a
15 datum (cons a _) = a
16
17 next : {a : Set} → Element a → Maybe (Element a)
18 next (cons _ n) = n
19
```

```
20 record SingleLinkedStack (a : Set) : Set where
21   field
22     top : Maybe (Element a)
23 open SingleLinkedStack
24
25 record Context : Set where
26   field
27     -- fields for concrete data segments
28     n       : ℕ
29     -- fields for stack
30     element : Maybe A
31
32
33
34
35
36 open import subtype Context as N
37
38 instance
39   ContextIsDataSegment : N.DataSegment Context
40   ContextIsDataSegment = record {get = (\c → c) ; set = (\_ c → c)}
41
42
43 record Meta  : Set1 where
44   field
45     -- context as set of data segments
46     context : Context
47     stack   : SingleLinkedStack A
48     nextCS  : N.CodeSegment Context Context
49
50
51
52
53 open import subtype Meta as M
54
55 instance
56   MetaIncludeContext : M.DataSegment Context
57   MetaIncludeContext = record { get = Meta.context
58                               ; set = (\m c → record m {context = c}) }
59
60   MetaIsMetaDataSegment : M.DataSegment Meta
61   MetaIsMetaDataSegment  = record { get = (\m → m) ; set = (\_ m → m) }
62
63
64 liftMeta : {X Y : Set} {{_ : M.DataSegment X}} {{_ : M.DataSegment Y}}
       → N.CodeSegment X Y → M.CodeSegment X Y
65 liftMeta (N.cs f) = M.cs f
66
67 liftContext : {X Y : Set} {{_ : N.DataSegment X}} {{_ : N.DataSegment Y}}
       → N.CodeSegment X Y → N.CodeSegment Context Context
68 liftContext {{x}} {{y}} (N.cs f) = N.cs (\c → N.DataSegment.set y c (f (
       N.DataSegment.get x c)))
69
70 -- definition based from Gears(209:5708390a9d88) src/parallel_execution
71
```

```
72 emptySingleLinkedStack : SingleLinkedStack A
73 emptySingleLinkedStack = record {top = nothing}
74
75
76 pushSingleLinkedStack : Meta → Meta
77 pushSingleLinkedStack m = M.exec (liftMeta n) (record m {stack = (push s
       e) })
78   where
79     n = Meta.nextCS m
80     s = Meta.stack  m
81     e = Context.element (Meta.context m)
82     push : SingleLinkedStack A → Maybe A → SingleLinkedStack A
83     push s nothing  = s
84     push s (just x) = record {top = just (cons x (top s))}
85
86
87
88 popSingleLinkedStack : Meta → Meta
89 popSingleLinkedStack m = M.exec (liftMeta n) (record m {stack = (st m) ;
       context = record con {element = (elem m)}})
90   where
91     n = Meta.nextCS m
92     con  = Meta.context m
93     elem : Meta → Maybe A
94     elem record {stack = record { top = (just (cons x _)) }} = just x
95     elem record {stack = record { top = nothing           }} = nothing
96     st : Meta → SingleLinkedStack A
97     st record {stack = record { top = (just (cons _ s)) }} = record {top
       = s}
98     st record {stack = record { top = nothing           }} = record {top
       = nothing}
99
100
101
102
103 pushSingleLinkedStackCS : M.CodeSegment Meta Meta
104 pushSingleLinkedStackCS = M.cs pushSingleLinkedStack
105
106 popSingleLinkedStackCS : M.CodeSegment Meta Meta
107 popSingleLinkedStackCS = M.cs popSingleLinkedStack
108
109
110 -- for sample
111
112 firstContext : Context
113 firstContext = record {element = nothing ; n = 0}
114
115
116 firstMeta : Meta
117 firstMeta = record { context = firstContext
118                    ; stack = emptySingleLinkedStack
119                    ; nextCS = (N.cs (\m → m))
120                    }
```

A-5　スタックの実装の検証

??節で取り上げたソースコードをリスト A.6 に示す。

リスト A.6: スタックの実装の検証コード (stack-subtype-sample.agda)

```
1  module stack-subtype-sample where
2
3  open import Level renaming (suc to S ; zero to O)
4  open import Function
5  open import Data.Nat
6  open import Data.Maybe
7  open import Relation.Binary.PropositionalEquality
8
9  open import stack-subtype ℕ
10 open import subtype Context  as N
11 open import subtype Meta     as M
12
13
14 record Num : Set where
15   field
16     num : ℕ
17
18 instance
19   NumIsNormalDataSegment : N.DataSegment Num
20   NumIsNormalDataSegment = record { get = (\c → record { num = Context.n
       c})
21                                    ; set = (\c n → record c {n = Num.num
       n})}
22   NumIsMetaDataSegment : M.DataSegment Num
23   NumIsMetaDataSegment = record { get = (\m → record {num = Context.n (
       Meta.context m)})
24                                 ; set = (\m n → record m {context =
       record (Meta.context m) {n = Num.num n}})}
25
26
27 plus3 : Num → Num
28 plus3 record { num = n } = record {num = n + 3}
29
30 plus3CS : N.CodeSegment Num Num
31 plus3CS = N.cs plus3
32
33
34
35 plus5AndPushWithPlus3 : {mc : Meta} {{_ : N.DataSegment Num}}
36                 → M.CodeSegment Num (Meta)
37 plus5AndPushWithPlus3 {mc} {{nn}} = M.cs (\n → record {context = con n ;
       nextCS = (liftContext {{nn}} {{nn}} plus3CS) ; stack = st} )
38   where
39     co    = Meta.context mc
40     con : Num → Context
41     con record { num = num } = N.DataSegment.set nn co record {num = num
       + 5}
42     st    = Meta.stack mc
```

```
43
44
45
46
47 push-sample : {{_ : N.DataSegment Num}} {{_ : M.DataSegment Num}} →
       Meta
48 push-sample {{nd}} {{md}} = M.exec {{md}} (plus5AndPushWithPlus3 {mc} {{
       nd}}) mc
49   where
50     con  = record { n = 4 ; element = just 0}
51     code = N.cs (\c → c)
52     mc   = record {context = con ; stack = emptySingleLinkedStack ;
       nextCS = code}
53
54
55 push-sample-equiv : push-sample ≡ record { nextCS  = liftContext plus3CS
56                                           ; stack   = record { top =
       nothing}
57                                           ; context = record { n = 9} }
58 push-sample-equiv = refl
59
60
61 pushed-sample : {m : Meta} {{_ : N.DataSegment Num}} {{_ : M.DataSegment
       Num}} →  Meta
62 pushed-sample {m} {{nd}} {{md}} = M.exec {{md}} (M.csComp {m} {{md}}
       pushSingleLinkedStackCS (plus5AndPushWithPlus3 {mc} {{nd}})) mc
63   where
64     con  = record { n = 4 ; element = just 0}
65     code = N.cs (\c → c)
66     mc   = record {context = con ; stack = emptySingleLinkedStack ;
       nextCS = code}
67
68
69
70 pushed-sample-equiv : {m : Meta} →
71                       pushed-sample {m} ≡ record { nextCS  = liftContext
       plus3CS
72                                                  ; stack   = record {
       top = just (cons 0 nothing) }
73                                                  ; context = record { n
         = 12} }
74 pushed-sample-equiv = refl
75
76
77
78 pushNum : N.CodeSegment Context Context
79 pushNum = N.cs pn
80   where
81     pn : Context → Context
82     pn record { n = n } = record { n = pred n  ; element = just n}
83
84
85 pushOnce : Meta → Meta
86 pushOnce m = M.exec pushSingleLinkedStackCS m
```

```
87
88 n-push : {m : Meta} {{_ : M.DataSegment Meta}} (n : ℕ) → M.CodeSegment
       Meta Meta
89 n-push {{mm}} (zero)      = M.cs {{mm}} {{mm}} id
90 n-push {m} {{mm}} (suc n) = M.cs {{mm}} {{mm}} (\m → M.exec {{mm}} {{mm
       }} (n-push {m} {{mm}} n) (pushOnce m))
91
92 popOnce : Meta → Meta
93 popOnce m = M.exec popSingleLinkedStackCS m
94
95 n-pop : {m : Meta} {{_ : M.DataSegment Meta}} (n : ℕ) → M.CodeSegment
       Meta Meta
96 n-pop {{mm}} (zero)      = M.cs {{mm}} {{mm}} id
97 n-pop {m} {{mm}} (suc n) = M.cs {{mm}} {{mm}} (\m → M.exec {{mm}} {{mm}}
       (n-pop {m} {{mm}} n) (popOnce m))
98
99
100
101 initMeta : ℕ  → Maybe ℕ → N.CodeSegment Context Context → Meta
102 initMeta n mn code = record { context = record { n = n ; element = mn}
103                           ; stack   = emptySingleLinkedStack
104                           ; nextCS  = code
105                           }
106
107 n-push-cs-exec = M.exec (n-push {meta} 3) meta
108   where
109     meta = (initMeta 5 (just 9) pushNum)
110
111
112 n-push-cs-exec-equiv : n-push-cs-exec ≡ record { nextCS  = pushNum
113                                                ; context = record {n = 2
       ; element = just 3}
114                                                ; stack   = record {top =
       just (cons 4 (just (cons 5 (just (cons 9 nothing)))))}}
115 n-push-cs-exec-equiv = refl
116
117
118 n-pop-cs-exec = M.exec (n-pop {meta} 4) meta
119   where
120     meta = record { nextCS  = N.cs id
121                   ; context = record { n = 0 ; element = nothing}
122                   ; stack   = record {top = just (cons 9 (just (cons 8 (
       just (cons 7 (just (cons 6 (just (cons 5 nothing))))))))))}
123                   }
124
125 n-pop-cs-exec-equiv : n-pop-cs-exec ≡ record { nextCS  = N.cs id
126                                              ; context = record { n = 0
       ; element = just 6}
127                                              ; stack   = record { top =
       just (cons 5 nothing)}
128                                              }
129
130 n-pop-cs-exec-equiv = refl
131
```

```
132
133 open ≡-Reasoning
134
135 id-meta : ℕ → ℕ → SingleLinkedStack ℕ → Meta
136 id-meta n e s = record { context = record {n = n ; element = just e}
137                        ; nextCS = (N.cs id) ; stack = s}
138
139 exec-comp : (f g : M.CodeSegment Meta Meta) (m : Meta) → M.exec (M.
       csComp {m} f g) m ≡ M.exec f (M.exec g m)
140 exec-comp (M.cs x) (M.cs _) m = refl
141
142
143 push-pop-type : ℕ → ℕ  → ℕ → Element ℕ → Set1
144 push-pop-type n e x s = M.exec (M.csComp {meta} (M.cs popOnce) (M.cs
       pushOnce)) meta ≡ meta
145   where
146     meta = id-meta n e record {top = just (cons x (just s))}
147
148 push-pop : (n e x : ℕ) → (s : Element ℕ) →  push-pop-type n e x s
149 push-pop n e x s = refl
150
151
152
153 pop-n-push-type : ℕ → ℕ → ℕ → SingleLinkedStack ℕ → Set1
154 pop-n-push-type n cn ce s = M.exec (M.csComp {meta} (M.cs popOnce) (n-
       push {meta} (suc n))) meta
155                           ≡ M.exec (n-push {meta} n) meta
156   where
157     meta = id-meta cn ce s
158
159 pop-n-push : (n cn ce : ℕ) → (s : SingleLinkedStack ℕ) → pop-n-push-
       type n cn ce s
160
161 pop-n-push zero cn ce s    = refl
162 pop-n-push (suc n) cn ce s = begin
163   M.exec (M.csComp {id-meta cn ce s} (M.cs popOnce) (n-push {id-meta cn
       ce (record {top = just (cons ce (SingleLinkedStack.top s))})} (suc (
       suc n)))) (id-meta cn ce s)
164   ≡⟨ refl ⟩
165   M.exec (M.csComp {id-meta cn ce s} (M.cs popOnce) (M.csComp {id-meta cn
        ce s} (n-push {id-meta cn ce (record {top = just (cons ce (
       SingleLinkedStack.top s))})} (suc n)) (M.cs pushOnce))) (id-meta cn ce
        s)
166   ≡⟨ exec-comp (M.cs popOnce) (M.csComp {id-meta cn ce s} (n-push {id-
       meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))})}
       (suc n)) (M.cs pushOnce)) (id-meta cn ce s) ⟩
167   M.exec (M.cs popOnce) (M.exec (M.csComp {id-meta cn ce s} (n-push {id-
       meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))})}
       (suc n)) (M.cs pushOnce)) (id-meta cn ce s))
168   ≡⟨ cong (\x → M.exec (M.cs popOnce) x) (exec-comp (n-push {id-meta cn
       ce (record {top = just (cons ce (SingleLinkedStack.top s))})} (suc n))
        (M.cs pushOnce) (id-meta cn ce s)) ⟩
169   M.exec (M.cs popOnce) (M.exec (n-push {id-meta cn ce (record {top =
```

```
    just (cons ce (SingleLinkedStack.top s))})} (suc n))(M.exec (M.cs
    pushOnce) (id-meta cn ce s)))
170 ≡⟨ refl ⟩
171 M.exec (M.cs popOnce) (M.exec (n-push {id-meta cn ce (record {top =
    just (cons ce (SingleLinkedStack.top s))})} (suc n)) (id-meta cn ce (
    record {top = just (cons ce (SingleLinkedStack.top s))})))
172 ≡⟨ sym (exec-comp (M.cs popOnce) (n-push {id-meta cn ce (record {top =
    just (cons ce (SingleLinkedStack.top s))})} (suc n)) (id-meta cn ce (
    record {top = just (cons ce (SingleLinkedStack.top s))}))) ⟩
173 M.exec (M.csComp {id-meta cn ce s} (M.cs popOnce) (n-push {id-meta cn
    ce (record {top = just (cons ce (SingleLinkedStack.top s))})} (suc n))
    ) (id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s
    ))})))
174 ≡⟨ pop-n-push n cn ce (record {top = just (cons ce (SingleLinkedStack.
    top s))}) ⟩
175 M.exec (n-push n) (id-meta cn ce (record {top = just (cons ce (
    SingleLinkedStack.top s))}))
176 ≡⟨ refl ⟩
177 M.exec (n-push n) (pushOnce (id-meta cn ce s))
178 ≡⟨ refl ⟩
179 M.exec (n-push n) (M.exec (M.cs pushOnce) (id-meta cn ce s))
180 ≡⟨ refl ⟩
181 M.exec (n-push {id-meta cn ce s} (suc n)) (id-meta cn ce s)
182 ■
183
184
185
186 n-push-pop-type : ℕ → ℕ → ℕ → SingleLinkedStack ℕ → Set1
187 n-push-pop-type n cn ce st = M.exec (M.csComp {meta} (n-pop {meta} n) (n-
    push {meta} n)) meta ≡ meta
188   where
189     meta = id-meta cn ce st
190
191 n-push-pop : (n cn ce : ℕ) → (s : SingleLinkedStack ℕ) → n-push-pop-
    type n cn ce s
192 n-push-pop zero    cn ce s = refl
193 n-push-pop (suc n) cn ce s = begin
194   M.exec (M.csComp {id-meta cn ce s} (n-pop {id-meta cn ce s}  (suc n)) (
    n-push {id-meta cn ce s} (suc n))) (id-meta cn ce s)
195   ≡⟨ refl ⟩
196   M.exec (M.csComp  {id-meta cn ce s}  (M.cs (\m → M.exec (n-pop {id-
    meta cn ce s} n) (popOnce m))) (n-push {id-meta cn ce s} (suc n))) (id
    -meta cn ce s)
197   ≡⟨ exec-comp (M.cs (\m → M.exec (n-pop n) (popOnce m))) (n-push {id-
    meta cn ce s}  (suc n))  (id-meta cn ce s) ⟩
198   M.exec (M.cs (\m → M.exec (n-pop  {id-meta cn ce s}  n) (popOnce m)))
    (M.exec (n-push {id-meta cn ce s} (suc n)) (id-meta cn ce s))
199   ≡⟨ refl ⟩
200   M.exec (n-pop n) (popOnce (M.exec (n-push {id-meta cn ce s} (suc n)) (
    id-meta cn ce s)))
201   ≡⟨ refl ⟩
202   M.exec (n-pop n) (M.exec (M.cs popOnce) (M.exec (n-push {id-meta cn ce
```

```
    s} (suc n)) (id-meta cn ce s)))
203 ≡⟨ cong (\x → M.exec (n-pop  {id-meta cn ce s}  n) x) (sym (exec-comp
    (M.cs popOnce) (n-push {id-meta cn ce s}  (suc n)) (id-meta cn ce s)))
     ⟩
204 M.exec (n-pop n) (M.exec (M.csComp {id-meta cn ce s} (M.cs popOnce) (n-
    push {id-meta cn ce s} (suc n))) (id-meta cn ce s))
205 ≡⟨ cong (\x → M.exec (n-pop {id-meta cn ce s}  n) x) (pop-n-push n cn
    ce s) ⟩
206 M.exec (n-pop n) (M.exec (n-push n) (id-meta cn ce s))
207 ≡⟨ sym (exec-comp (n-pop n) (n-push n) (id-meta cn ce s)) ⟩
208 M.exec (M.csComp (n-pop n) (n-push n)) (id-meta cn ce s)
209 ≡⟨ n-push-pop n cn ce s ⟩
210 id-meta cn ce s
211 ■
```