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

Abstract

目 次

図 目 次

表 目 次

リスト目次

第1章　Agda における Continuation based C の表現

CbC の項を部分型を用いて Agda 上に記述していく。DataSegment と CodeSegment の定義、CodeSegment の接続と実行、メタ計算を定義し、 Agda 上で実行できることを確認する。また、Agda 上で定義した DataSegment とそれに付随する CodeSegment の持つ性質を Agda 上で証明していく。

1.1　DataSegment の定義

まず DataSegment から定義していく。DataSegment はレコード型で表現できるため、Agda のレコードをそのまま利用できる。例えは **??** に示していた a と b を加算して c を出力するプログラムに必要な DataSegment を記述すると 1.1 のようになる。cs0 は a と b の二つの Int 型の変数を利用するため、対応する ds0 は a と b のフィールドを持つ。cs1 は計算結果を格納する c という名前の変数のみを持つので、同様に ds1 も c のみを持つ。

リスト 1.1: Agda における DataSegment の定義

```
1 record ds0 : Set where
2   field
3     a : Int
4     b : Int
5
6 record ds1 : Set where
7   field
8     c : Int
```

1.2　CodeSegment の定義

次に CodeSegment を定義する。CodeSegment は DataSegment を取って DataSegment を返すものである。よって $I \to O$ を内包するデータ型を定義する。

レコード型の型は Set なので、Set 型を持つ変数 I と O を型変数に持ったデータ型 CodeSegment を定義する。I は Input DataSegment の型であり、 O は Output DataSegment である。

CodeSegment 型のコンストラクタには cs があり、Input DataSegment を取って Output DataSegment を返す関数を取る。具体的なデータ型の定義はリスト 1.2 のようになる。

リスト 1.2: Agda における CodeSegment 型の定義

```
1 data CodeSegment {l1 l2 : Level} (I : Set l1) (O : Set l2) : Set (l ⊔ l1
    ⊔ l2) where
2   cs : (I → O) → CodeSegment I O
```

この CodeSegment 型を用いて CodeSegment の処理本体を記述する。

まず計算の本体となる cs0 に注目する。cs0 は二つの Int 型変数を持つ ds0 を取り、一つの Int 型変数を作った上で cs1 に軽量継続を行なう。DataSegment はレコードなので、a と b のフィールドから値を取り出した上で加算を行ない、c を持つレコードを生成する。そのレコードを引き連れたまま cs1 へと goto する。

次に cs1 に注目する。cs1 は値に触れず cs2 へと goto するだけである。よって何もせずにそのまま goto する関数をコンストラクタ cs に渡すだけで良い。

最後に cs2 である。cs2 はリスト **??**では省略していたが、今回は計算を終了させる CodeSegment として定義する。どの CodeSegment にも軽量継続せずに値を持ったまま計算を終了させる。コンストラクタ cs には関数を与えなくては値を構成できないため、何もしない関数である id を渡している。

最後に計算をする cs0 へと軽量継続する main を定義する。例として、 a の値を 100 とし、 b の値を 50 としている。

cs0, cs1, cs2, main を Agda で定義するとリスト 1.3 のようになる。

リスト 1.3: Agda における CodeSegment の定義

```
1  cs2 : CodeSegment ds1 ds1
2  cs2 = cs id
3
4  cs1 : CodeSegment ds1 ds1
5  cs1 = cs (\d -> goto cs2 d)
6
7  cs0 : CodeSegment ds0 ds1
8  cs0 = cs (\d -> goto cs1 (record {c = (ds0.a d) + (ds0.b d)}))
9
10 main : ds1
11 main = goto cs0 (record {a = 100 ; b = 50})
```

正しく計算が行なえたなら値 150 が得られるはずである。

1.3　ノーマルレベル計算の実行

プログラムを実行することは goto を定義することと同義である。軽量継続 goto の性質としては

- 次に実行する CodeSegment を指定する
- CodeSegment に渡すべき DataSegment を指定する
- 現在実行している CodeSegment から制御を指定された CodeSegment へと移動させる

がある。Agda における CodeSegment の本体は関数である。関数をそのまま使用すると再帰を許してしまうために CbC との対応が失われてしまう。よって、goto を利用できるのは関数の末尾のみである、という制約を関数に付け加える必要がある。

この制約さえ満たせば、CodeSegment の実行は CodeSegment 型から関数本体を取り出し、レコード型を持つ値を適用することに相当する。具体的に goto を関数として適用するとリスト 1.4 のようになる。

リスト 1.4: Agda における goto の定義

```
1 goto : {l1 l2 : Level} {I : Set l1} {O : Set l2}
2    -> CodeSegment I O -> I -> O
3 goto (cs b) i = b i
```

この goto の定義を用いることで main などの関数が評価できるようになり、値 150 が得られる。本文中での CodeSegment の定義は一部を抜粋している。実行可能な Agda のソースコードは付録に載せる。

1.4　Meta DataSegment の定義

ノーマルレベルの CbC を Agda 上で記述し、実行することができた。次にメタレベルの計算を Agda 上で記述していく。

Meta DataSegment はノーマルレベルの DataSegment の集合として定義できるものであり、全ての DataSegment の部分型であった。ノーマルレベルの DataSegment はプログラムによって変更されるので、事前に定義できるものではない。ここで、Agda の Parameterized Module を利用して、「Meta DataSegment の上位型は DataSegment である」のように DataSegment を定義する。こうすることにより、全てのプログラムは一つ以上の Meta DataSegment を持ち、任意の個数の DataSegment を持つ。また、Meta DataSegment をメタレベルの DataSegment として扱うことにより、「Meta DataSegment

の部分型である Meta Meta DataSegment」を定義できるようになる。階層構造でメタレベルを表現することにより、計算の拡張を自在に行なうことができる。

具体的な Meta DataSegment の定義はリスト 1.5 のようになる。型システム `subtype` は、Meta DataSegment である `Context` を受けとることにより構築される。Context を Meta DataSegment とするプログラム上では DataSegment は Meta CodeSegment の上位型となる。その制約を `DataSegment` 型は表わしている。

リスト 1.5: Agda における Meta DataSegment の定義

```
1 module subtype {l : Level} (Context : Set l) where
2
3 record DataSegment {ll : Level} (A : Set ll) : Set (l ⊔ ll) where
4   field
5     get : Context → A
6     set : Context → A → Context
```

ここで、関数を部分型に拡張する S-ARROW をもう一度示す。

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \text{S-ARROW}$$

S-ARROW は、前提である部分型関係 $T_1 <: S_1$ と $S_2 <: T_2$ が成り立つ時に、 上位型 $S_1 \to S_2$ の関数を、部分型 $T_1 \to T_2$ に拡張できた。ここでの上位型は DataSegment であり、部分型は Meta DataSegment である。制約 `DataSegment` の `get` は、 Meta DataSegment から DataSegment が生成できることを表す。これは前提 $T_1 <: S_1$ に相当する。そして、`set` は $S_2 <: T_2$ に相当する。しかし、任意の DataSegment が Meta DataSegment の部分型となるには、 DataSegment が Meta DataSegment よりも多くの情報を必ず持たなくてはならないが、これは通常では成り立たない。だが、メタ計算を行なう際には常に Meta DataSegment を一つ以上持っていると仮定するならば成り立つ。実際、GearsOS における赤黒木では Meta DataSegment に相当する `Context` を常に持ち歩いている。GearsOS における計算結果はその持ち歩いている Meta DataSegment の更新に相当するため、常に Meta DataSegment を引き連れていることを無視すれば DataSegment から Meta DataSegment を導出できる。よって $S_2 <: T_2$ が成り立つ。

なお、 $S_2 <: T_2$ は Output DataSegment を Meta DataSegment を格納する作業に相当し、 $T_1 <: S_1$ は Meta DataSegment から Input DataSegment を取り出す作業であるため、これは明らかに `stub` である。

1.5 Meta CodeSegment の定義

Meta DataSegment が定義できたので Meta CodeSegment を定義する。実際、DataSegment が Meta DataSegment に拡張できたため、Meta CodeSegment の定義には比較的変

更は無い。ノーマルレベルの CodeSegment 型に、DataSegment を取って DataSegment を返す、という制約を明示的に付けるだけである (リスト 1.6)

リスト 1.6: Agda における Meta CodeSegment の定義

```
1 data CodeSegment {l1 l2 : Level} (A : Set l1) (B : Set l2) : Set (l ⊔ l1
    ⊔ l2) where
2   cs : {{_ : DataSegment A}} {{_ : DataSegment B}}
3         → (A → B) → CodeSegment A B
```

1.6　メタレベル計算の実行

Meta DataSegment と Meta CodeSegment の定義を行なったので、残るは実行である。

実行はノーマルレベルにおいては軽量継続 goto を定義することによって表せた。メタレベル実行ではそれを Meta CodeSegment と Meta DataSegment を扱えるように拡張する。Meta DataSegment は Parameterized Module の引数 Context に相当するため、Meta CodeSegment は Context を取って Context を返す CodeSegment となる。軽量継続 goto と区別するために名前を exec とするリスト 1.7 のように定義できる。行なっていることは Meta CodeSegment の本体部分に Meta DataSegment を渡す、という goto と変わらないが、set と get を用いることで上位型である任意の DataSegment を実行する CodeSegment も Meta CodeSegment として一様に実行できる。

リスト 1.7: Agda におけるメタレベル実行の定義

```
1 exec : {l1 l2 : Level} {I : Set l1} {O : Set l2}
2        {{_ : DataSegment I}} {{_ : DataSegment O}}
3        → CodeSegment I O → Context → Context
4 exec {l} {{i}} {{o}}  (cs b) c = set o c (b (get i c))
```

実行例として、リスト **??** に示していた a と b の値を加算して c に代入するプログラムを考える。実行する際に c の値を c' に保存してから加算ようなメタ計算を考える。c の値を c' に保存するタイミングは軽量継続時にユーザが指定する。よって軽量継続を行なうのと同等の情報を保持してなくてはならない。そのために Meta Meta DataSegment Meta には制御を移す対象であるノーマルレベル CodeSegment を持つ。値を格納する c' の位置は Meta DataSegment でも Meta Meta DataSegment でも構わないが、今回は Meta Meta DataSegemnt に格納するものとする。それらを踏まえた上での Meta Meta DataSegment の Agda 上での定義は 1.8 のようになる。なお、goto などの名前の衝突を避けるためにノーマルレベルの定義は N に、メタレベルの定義は M へと名前を付けかえている。

リスト 1.8: Agda における Meta Meta DataSegment の定義例

```
1 ...
2 open import subtype Context as N
```

```
3
4  record Meta : Set where
5    field
6      context : Context
7      c'      : Int
8      next    : N.CodeSegment Context Context
9
10 open import subtype Meta as M
11 ...
```

定義した Meta を利用して、c を c' に保存するメタ計算 push を定義する。より構文が CbC に似るように gotoMeta を糖衣構文的に定義する。gotoMeta や push で利用している liftContext や liftMeta はノーマルレベル計算をメタ計算レベルとするように型を明示的に変更するものである。結果的に main の goto を gotoMeta に置き換えることにより、c の値を計算しながら保存できる。リスト 1.9 に示したプログラムでは、通常レベルのコードセグメントを全く変更せずにメタ計算を含む形に拡張している。加算を行なう前の c の値が 70 であったとした時、計算結果 150 は c に格納されるが、c' には 70 に保存されている。

リスト 1.9: Agda における Meta Meta CodeSegment の定義と実行例

```
1  ...
2  -- meta level
3  liftContext : {X Y : Set} {{_ : N.DataSegment X}} {{_ : N.DataSegment Y}}
       -> N.CodeSegment X Y -> N.CodeSegment Context Context
4  liftContext {{x}} {{y}} (N.cs f) = N.cs (\c -> N.DataSegment.set y c (f (
       N.DataSegment.get x c)))
5
6  liftMeta : {X Y : Set} {{_ : M.DataSegment X}} {{_ : M.DataSegment Y}} ->
       N.CodeSegment X Y -> M.CodeSegment X Y
7  liftMeta (N.cs f) = M.cs f
8
9  gotoMeta : {I O : Set} {{_ : N.DataSegment I}} {{_ : N.DataSegment O}} ->
       M.CodeSegment Meta Meta -> N.CodeSegment I O -> Meta -> Meta
10 gotoMeta mCode code m = M.exec mCode (record m {next = (liftContext code)
       })
11
12 push : M.CodeSegment Meta Meta
13 push = M.cs (\m -> M.exec (liftMeta (Meta.next m)) (record m {c' =
       Context.c (Meta.context m)}))
14
15 -- normal level
16
17 cs2 : N.CodeSegment ds1 ds1
18 cs2 = N.cs id
19
20 cs1 : N.CodeSegment ds1 ds1
21 cs1 = N.cs (\d -> N.goto cs2 d)
22
23 cs0 : N.CodeSegment ds0 ds1
24 cs0 = N.cs (\d -> N.goto cs1 (record {c = (ds0.a d) + (ds0.b d)}))
25
```

```
26 -- meta level (with extended normal)
27 main : Meta
28 main = gotoMeta push cs0 (record {context = (record {a = 100 ; b = 50 ; c
       = 70}) ; c' = 0 ; next = (N.cs id)})
29 -- record {context = record {a = 100 ; b = 50 ; c = 150} ; c' = 70 ; next
       = (N.cs id)}
```

なお、CbC におけるメタ計算を含む軽量継続 `goto meta` と Agda におけるメタ計算実行の比較はリスト **??**のようになる

CodeSegment や Meta CodeSegment などの定義が多かったため、どの処理やデータがどのレベルに属するか複雑になったため、一度図にてまとめる。Meta DataSegment を含む任意の DataSegment は Meta DataSegment になりえるので、この階層構造は任意の段数定義することが可能である。

1.7　Agda を用いた Continuation based C の検証

Agda において CbC の CodeSegment と DataSegment を定義することができた。実際の CbC のコードを Agda に変換し、それらの性質を証明していく。

ここでは GearsOS が持つ DataSegment `SingleLinkedStack` を証明していく。この `SingleLinkedStack` はポインタを利用した片方向リストを用いて実装されている。

CbC における DataSegment `SingleLinkedStack` の定義はリスト 1.10 のようになっている。

リスト 1.10: CbC における構造体 stack の定義

```
1 struct SingleLinkedStack {
2     struct Element* top;
3 } SingleLinkedStack;
4 struct Element {
5     union Data* data;
6     struct Element* next;
7 } Element;
```

次に Agda における `SingleLinkedStack` の定義について触れるが、Agda にはポインタが無いため、メモリ確保や `NULL` の定義は存在しない。CbC におけるメモリ確保部分はノーマルレベルには出現しないと仮定し、`NULL` の表現には Agda の標準ライブラリに存在する `Data.Maybe` を用いる。

`Data.Maybe` の `maybe` 型は、コンストラクタを二つ持つ。片方のコンストラクタ `just` は値を持ったデータであり、ポインタの先に値があることに対応している。一方のコンストラクタ `nothing` は値を持たない。これは値が存在せず、ポインタの先が確保されていない (`NULL` ポインタである) ことを表現できる。

リスト 1.11: Agda における Maybe の定義

```
1 data Maybe {a} (A : Set a) : Set a where
2   just    : (x : A) → Maybe A
3   nothing : Maybe A
```

Maybe を用いて片方向リストを Agda 上に定義するとリスト 1.12 のようになる。CbC とほぼ同様の定義ができている。CbC、Agda 共に SingleLinkedStack は Element 型の top を持っている。Element 型は値と次の Element を持つ。CbC ではポインタで表現していた部分が Agda では Maybe で表現されているが、Element 型の持つ情報は変わっていない。

リスト 1.12: Agda における片方向リストを用いたスタックの定義

```
1  data Element (a : Set) : Set where
2    cons : a -> Maybe (Element a) -> Element a
3
4  datum : {a : Set} -> Element a -> a
5  datum (cons a _) = a
6
7  next : {a : Set} -> Element a -> Maybe (Element a)
8  next (cons _ n) = n
9
10 record SingleLinkedStack (a : Set) : Set where
11   field
12     top : Maybe (Element a)
```

Agda で片方向リストを利用する DataSegment の定義をリスト **??**に示す。ノーマルレベルからアクセス可能な場所として Context に element フィールドを追加する。そしてノーマルレベルからアクセスできないよう分離した Meta Meta DataSegment である Meta にスタックの本体を格納する。CbC における実装では ... で不定であった next も、agda ではメタレベルのコードセグメント nextCS となり、きちんと型付けされている。

リスト 1.13: スタックを利用するための DataSegment の定義

```
1  record Context : Set where
2    field
3      -- fields for stack
4      element : Maybe A
5
6
7  open import subtype Context as N
8
9  record Meta  : Set^^e2^^82^^81 where
10   field
11     -- context as set of data segments
12     context : Context
13     stack   : SingleLinkedStack A
14     nextCS  : N.CodeSegment Context Context
15
16 open import subtype Meta as M
```

次にスタックへの操作に注目する。スタックへと値を積む pushSingleLinkedStack と値を取り出す popSingleLinkedStack の CbC 実装をリスト 1.14 に示す。SingleLinkedStack は Meta CodeSegment であり、メタ計算を実行した後には通常の CodeSegment へと操作を移す。そのために next という名前で次のコードセグメントを保持している。具体的な next はコンパイル時にしか分からないため、... 構文を用いて不定としている。

pushSingleLinkedStack は element を新しく確保し、値を入れた後に next へと繋げ、 top を更新して軽量継続する。popSingleLinkedStack は先頭が空でなければ先頭の値を top から取得し、element を一つ進める。値が空であれば data を NULL にしたまま軽量継続を行なう。

リスト 1.14: CbC における SingleLinkedStack を操作する Meta CodeSegment

```
1  __code pushSingleLinkedStack(struct SingleLinkedStack* stack,union Data*
       data, __code next(...)) {
2      Element* element = new Element();
3      element->next = stack->top;
4      element->data = data;
5      stack->top = element;
6      goto next(...);
7  }
8
9  __code popSingleLinkedStack(struct SingleLinkedStack* stack, __code next(
       union Data* data, ...)) {
10     if (stack->top) {
11         data = stack->top->data;
12         stack->top = stack->top->next;
13     } else {
14         data = NULL;
15     }
16     goto next(data, ...);
17 }
```

次に Agda における定義をリスト 1.15 に示す。同様に pushSingleLinkedStack と popSingleLinkedStack を定義している。pushsinglelinkedstack では、スタックの値を更新したのちにノーマルレベルの CodeSegment である n を exec している。なお、liftMeta はノーマルレベルの計算をメタレベルとする関数である。

実際に値を追加する部分は where 句に定義された関数 push である。これはスタックへと積む値が空であれば追加を行なわず、値がある時は新たに element を作成して top を更新している。本来の CbC の実装では空かチェックはしていないが、値が空であるかに関わらずにスタックに積んでいるために挙動は同じである。

次に popSingleLinkedStack に注目する。こちらも操作後に nextCS へと継続を移すようになっている。

実際に値を取り出す操作はノーマルレベルからアクセスできる element の値の確定と、アクセスできない stack の更新である。

element については、 top が空なら取り出した後の値は無いので element は nothing である。top が空でなければ element は top の値となる。

stack は空なら空のままであり、top に値があればその先頭を捨てる。ここで、pop の実装はスタックが空であっても、例外を送出したり停止したりせずに処理を続行できることが分かる。

リスト 1.15: Agda における片方向リストを用いたスタックの定義

```
pushSingleLinkedStack : Meta -> Meta
pushSingleLinkedStack m = M.exec (liftMeta n) (record m {stack = (push s e) })
  where
    n = Meta.nextCS m
    s = Meta.stack  m
    e = Context.element (Meta.context m)
    push : SingleLinkedStack A -> Maybe A -> SingleLinkedStack A
    push s nothing  = s
    push s (just x) = record {top = just (cons x (top s))}

popSingleLinkedStack : Meta -> Meta
popSingleLinkedStack m = M.exec (liftMeta n) (record m {stack = (st m) ; context = record con {element = (elem m)}})
  where
    n = Meta.nextCS m
    con  = Meta.context m
    elem : Meta -> Maybe A
    elem record {stack = record { top = (just (cons x _)) }} = just x
    elem record {stack = record { top = nothing           }} = nothing
    st : Meta -> SingleLinkedStack A
    st record {stack = record { top = (just (cons _ s)) }} = record {top = s}
    st record {stack = record { top = nothing           }} = record {top = nothing}


pushSingleLinkedStackCS : M.CodeSegment Meta Meta
pushSingleLinkedStackCS = M.cs pushSingleLinkedStack

popSingleLinkedStackCS : M.CodeSegment Meta Meta
popSingleLinkedStackCS = M.cs popSingleLinkedStack
```

また、この章で取り上げた CbC と Agda の動作するソースコードは付録に載せる。

1.8　スタックの実装の検証

定義した SingleLinkedStack に対して証明を行なっていく。ここでの証明は SingleLinkedStack の処理が特定の性質を持つことを保証することである。

例えば

- スタックに積んだ値は取り出せる
- 値は複数積むことができる
- スタックから値を取り出すことができる
- スタックから取り出す値は積んだ値である
- スタックから値を取り出したらその値は無くなる
- スタックに値を積んで取り出すとスタックの内容は変わらない

といった多くの性質がある。

ここでは、最後に示した「スタックに値を積んで取り出すとスタックの内容は変わらない」ことについて示していく。この性質を具体的に書き下すと以下のようになる。

定義 **1.1** 任意のスタック s に対して

- 任意の値 n
- 値 x を積む操作 push(x, s)
- 値を一つスタックから取り出す操作 pop(s)

がある時、

pop . push(n) s = s

である。

これを Agda 上で定義するとリスト 1.16 のようになる。Agda 上の定義ではスタックそのものではなく、スタックを含む任意の `Meta` に対してこの性質を証明する。この定義が `Meta` の値によらず成り立つことを、自然数の加算の交換法則と同様に等式変形を用いて証明していく。

リスト 1.16: Agda におけるスタックの性質の定義 (1)

```
1 pushOnce : Meta → Meta
2 pushOnce m = M.exec pushSingleLinkedStackCS m
3
4 popOnce : Meta → Meta
5 popOnce m = M.exec popSingleLinkedStackCS m
6
7 push-pop-type : Meta → Set1
8 push-pop-type meta =
9     M.exec (M.csComp  (M.cs popOnce) (M.cs pushOnce)) meta ≡ meta
```

今回注目する条件分けは、スタック本体である `stack` と、push や pop を行なうための値を格納する `element` である。それぞれが持ち得る値を場合分けして考えていく。

- スタックが空である場合
  - element が存在する場合

    値が存在するため、 push は実行される。push が実行されたためスタックに値があるため、pop が成功する。pop が成功した結果スタックは空となるため元のスタックと同一となり成り立つ。
  - element が存在しない場合

    値が存在しないため、 push が実行されない。push が実行されなかったため、スタックは空のままであり、pop も実行されない。結果スタックは空のままであり、元のスタックと一致する。
- スタックが空でない場合
  - element が存在する場合

    element に設定された値 n が push され、スタックに一つ値が積まれる。スタックの先頭は n であるため、pop が実行されて n は無くなる。結果、スタックは実行する前の状態に戻る。
  - element が存在しない場合

    element に値が存在しないため、push は実行されない。スタックは空ではないため、pop が実行され、先頭の値が無くなる。実行後、スタックは一つ値を失なっているため、これは成りたたない。

スタックが空でなく、push する値が存在しないときにこの性質は成り立たないことが分かった。また、`element` が空でない制約を仮定に加えることでこの命題は成り立つようになる。

push 操作と pop 操作を連続して行なうとスタックが元に復元されることは分かった。ここで SingleLinkedStack よりも範囲を広げて `Meta` も復元されるかを考える。一見これも自明に成り立ちそうだが、push 操作と pop 操作は操作後に実行される CodeSegment を持っている。この CodeSegment は任意に設定できるため、`Meta` 内部の DataSegement が書き換えられる可能性がある。よってこれも制約無しでは成り立たない。

逆にいえば、CodeSegment を指定してしまえば `Meta` に関しても push/pop の影響を受けないことを保証できる。全く値を変更しない CodeSegment `id` を指定した際には自明にこの性質が導ける。実際、 Agda 上でも等式変形を明示的に指定せず、定義からの推論でこの証明を導ける (リスト 1.17)。

なお、今回 SingleLinkedStack に積むことができる値は Agda の標準ライブラリ (Data.Nat) における自然数型 ℕ としている。push/pop 操作の後の継続が Meta に影響を与えない制約は id-meta に表れている。これは Meta を構成する要素を受け取り、継続先の Code-Segment に恒等関数 id を指定している。加えて、スタックが空で無い制約 where 句の meta に表れている。必ずスタックの先頭 top には値 x が入っており、それ以降の値は任意としている。よってスタックは必ず一つ以上の値を持ち、空でないという制約を表わせる。証明は refl によって与えられる。つまり定義から自明に推論可能となっている。

リスト 1.17: Agda におけるスタックの性質の証明 (1)

```
1  id-meta : ℕ → ℕ → SingleLinkedStack ℕ → Meta
2  id-meta n e s = record { context = record {n = n ; element = just e}
3                         ; nextCS = (N.cs id) ; stack = s}
4
5  push-pop-type : ℕ → ℕ  → ℕ → Element ℕ → Set1
6  push-pop-type n e x s = M.exec (M.csComp {meta} (M.cs popOnce) (M.cs
       pushOnce)) meta ≡ meta
7    where
8      meta = id-meta n e record {top = just (cons x (just s))}
9
10 push-pop : (n e x : ℕ) → (s : Element ℕ) →  push-pop-type n e x s
11 push-pop n e x s = refl
```

ここで興味深い点は、 SingleLinkedStack の実装から証明に仮定が必要なことが証明途中でに分かった点にある。例えば、CbC の SingleLinkedStack 実装の push/pop 操作は失敗しても成功しても指定された CodeSegment に軽量継続する。この性質により、指定された CodeSegment によっては、スタックの操作に関係なく Meta の内部の DataSegemnt が書き換えられる可能性があることが分かった。スタックの操作の際に行なわれる軽量継続の利用方法は複数考えられる。例えば、スタックが空である際に pop を行なった時はエラー処理用の継続を行なう、といった実装も可能である。実装が異なれば、同様の性質でも証明は異なるものとなる。このように、実装そのものを適切に型システムで定義できれば、明示されていない実装依存の仕様も証明時に確定させることができる。

証明した定理をより一般的な「任意の自然数回だけスタックへ値を積み、その後同じ回数スタックから値を取り出すとスタックは操作前と変わらない」という形に拡張する。この性質を Agda で定義するとリスト 1.18 のようになる。自然数 n 回だけ push/pop することを記述するために Agda 上に n-push 関数と n-pop 関数を定義している。それぞれ一度操作を行なった後に再帰的に自身を呼び出す再帰関数である。

リスト 1.18: Agda におけるスタックの性質の定義 (2)

```
1  n-push : {m : Meta} {{_ : M.DataSegment Meta}} (n : ℕ) → M.CodeSegment
       Meta Meta
2  n-push {{mm}} (zero)      = M.cs {{mm}} {{mm}} id
3  n-push {m} {{mm}} (suc n) = M.cs {{mm}} {{mm}} (\m → M.exec {{mm}} {{mm
       }} (n-push {m} {{mm}} n) (pushOnce m))
```

```
4
5  n-pop : {m : Meta} {{_ : M.DataSegment Meta}} (n : ℕ) → M.CodeSegment
       Meta Meta
6  n-pop {{mm}} (zero)      = M.cs {{mm}} {{mm}} id
7  n-pop {m} {{mm}} (suc n) = M.cs {{mm}} {{mm}} (\m → M.exec {{mm}} {{mm}}
       (n-pop {m} {{mm}} n) (popOnce m))
8
9  pop-n-push-type : ℕ → ℕ → ℕ → SingleLinkedStack ℕ → Set1
10 pop-n-push-type n cn ce s = M.exec (M.csComp {meta} (M.cs popOnce) (n-
       push {meta} (suc n))) meta
11                           ≡ M.exec (n-push {meta} n) meta
12   where
13     meta = id-meta cn ce s
```

この性質の証明は少々複雑である。結論から先に示すとリスト 1.19 のように証明できる。

リスト 1.19: Agda におけるスタックの性質の証明 (2)

```
1  pop-n-push-type : ℕ → ℕ → ℕ → SingleLinkedStack ℕ → Set1
2  pop-n-push-type n cn ce s = M.exec (M.csComp (M.cs popOnce) (n-push (suc
       n))) meta
3                            ≡ M.exec (n-push n) meta
4    where
5      meta = id-meta cn ce s
6
7  pop-n-push : (n cn ce : ℕ) → (s : SingleLinkedStack ℕ) → pop-n-push-
       type n cn ce s
8  pop-n-push zero cn ce s    = refl
9  pop-n-push (suc n) cn ce s = begin
10   M.exec (M.csComp  (M.cs popOnce) (n-push  (suc (suc n)))) (id-meta cn
       ce s)
11   ≡⟨ refl ⟩
12   M.exec (M.csComp (M.cs popOnce) (M.csComp  (n-push  (suc n)) (M.cs
       pushOnce))) (id-meta cn ce s)
13   ≡⟨ exec-comp (M.cs popOnce) (M.csComp  (n-push  (suc n)) (M.cs pushOnce
       )) (id-meta cn ce s) ⟩
14   M.exec (M.cs popOnce) (M.exec (M.csComp  (n-push  (suc n)) (M.cs
       pushOnce)) (id-meta cn ce s))
15   ≡⟨ cong (\x → M.exec (M.cs popOnce) x) (exec-comp (n-push (suc n)) (M.
       cs pushOnce) (id-meta cn ce s)) ⟩
16   M.exec (M.cs popOnce) (M.exec (n-push  (suc n))(M.exec (M.cs pushOnce)
       (id-meta cn ce s)))
17   ≡⟨ refl ⟩
18   M.exec (M.cs popOnce) (M.exec (n-push  (suc n)) (id-meta cn ce (record
       {top = just (cons ce (SingleLinkedStack.top s))})))
19   ≡⟨ sym (exec-comp (M.cs popOnce) (n-push (suc n)) (id-meta cn ce (
       record {top = just (cons ce (SingleLinkedStack.top s))}))) ⟩
20   M.exec (M.csComp  (M.cs popOnce) (n-push  (suc n))) (id-meta cn ce (
       record {top = just (cons ce (SingleLinkedStack.top s))}))
21   ≡⟨ pop-n-push n cn ce (record {top = just (cons ce (SingleLinkedStack.
       top s))}) ⟩
```

```
22    M.exec (n-push n) (id-meta cn ce (record {top = just (cons ce (
        SingleLinkedStack.top s))}))
23    ≡⟨ refl ⟩
24    M.exec (n-push n) (pushOnce (id-meta cn ce s))
25    ≡⟨ refl ⟩
26    M.exec (n-push n) (M.exec (M.cs pushOnce) (id-meta cn ce s))
27    ≡⟨ refl ⟩
28    M.exec (n-push (suc n)) (id-meta cn ce s)
29    ■
30
31
32
33  n-push-pop-type : ℕ →  ℕ  → ℕ → SingleLinkedStack ℕ → Set1
34  n-push-pop-type n cn ce st = M.exec (M.csComp (n-pop n) (n-push n)) meta
        ≡ meta
35    where
36      meta = id-meta cn ce st
37
38  n-push-pop : (n cn ce : ℕ) → (s : SingleLinkedStack ℕ) → n-push-pop-
        type n cn ce s
39  n-push-pop zero    cn ce s = refl
40  n-push-pop (suc n) cn ce s = begin
41    M.exec (M.csComp (n-pop (suc n)) (n-push (suc n))) (id-meta cn ce s)
42    ≡⟨ refl ⟩
43    M.exec (M.csComp (M.cs (\m → M.exec (n-pop n) (popOnce m))) (n-push (
        suc n))) (id-meta cn ce s)
44    ≡⟨ exec-comp (M.cs (\m → M.exec (n-pop n) (popOnce m))) (n-push (suc n
        ))  (id-meta cn ce s) ⟩
45    M.exec (M.cs (\m → M.exec (n-pop  n) (popOnce m))) (M.exec (n-push (
        suc n)) (id-meta cn ce s))
46    ≡⟨ refl ⟩
47    M.exec (n-pop n) (popOnce (M.exec (n-push (suc n)) (id-meta cn ce s)))
48    ≡⟨ refl ⟩
49    M.exec (n-pop n) (M.exec (M.cs popOnce) (M.exec (n-push (suc n)) (id-
        meta cn ce s)))
50    ≡⟨ cong (\x → M.exec (n-pop n) x) (sym (exec-comp (M.cs popOnce) (n-
        push (suc n)) (id-meta cn ce s))) ⟩
51    M.exec (n-pop n) (M.exec (M.csComp (M.cs popOnce) (n-push (suc n))) (id
        -meta cn ce s))
52    ≡⟨ cong (\x → M.exec (n-pop n) x) (pop-n-push n cn ce s) ⟩
53    M.exec (n-pop n) (M.exec (n-push n) (id-meta cn ce s))
54    ≡⟨ sym (exec-comp (n-pop n) (n-push n) (id-meta cn ce s)) ⟩
55    M.exec (M.csComp (n-pop n) (n-push n)) (id-meta cn ce s)
56    ≡⟨ n-push-pop n cn ce s ⟩
57    id-meta cn ce s
58    ■
```

これは以下のような形の証明になっている。

- 「n 回 push した後に n 回 pop しても同様になる」という定理を n-push-pop とおく。
- n-push-pop は自然数 n と特定の Meta に対して exec (n-pop (suc n)) . (n-push (suc n))

が成り立つことである

- 特定の Meta とは、 push/pop 操作の後の継続が DataSegment を変更しない Meta である。
- また、簡略化のために csComp による CodeSegment の合成を二項演算子 . とおく
  - 例えば exec (csComp f g) x は exec (f . g) x となる。
- n-push-pop を証明するための補題 pop-n-push を定義する
- n-push-pop とは「n+1 回 push して 1 回 pop することは、n 回 push することと等しい」という補題である。
- n-push-pop は exec (pop . n-push (suc n)) m = exec (n-push n) m と表現できる。
- n-push-pop の n が zero の時は直ちに成り立つ。
- n-push-pop の n が zero でない時 (suc n である時) は以下のように証明できる。
  - exec (n-push (suc n)) m を X とおく
  - exec (pop . n-push (suc (suc n))) m = X
  - n-push の定義より exec (pop . (n-push (suc n) . push)) m = X
  - 補題 exec-comp より exec (pop (exec (n-push (suc n) . push) m)) = X
  - 補題 exec-comp より exec (pop (exec (n-push (suc n) (exec push m)))) = X
  - 一度 push した結果を m' とおくと exec (pop (exec (n-push (suc n) m'))) = X
  - n-push-pop より exec (exec (n-push n m')) = X
  - push の定義より exec (exec (n-push n (exec push m))) = X
  - n-push の定義より exec (exec (n-push (suc n) m) = X となる
  - 全く同一の項に変更できたので証明終了
- 次に n-push-pop の証明を示す。
- n-push-pop の n が zero の時は、 suc zero 回の push/pop が行なわれるため、push-pop より成り立つ。
- n-push-pop の n が zero でない時は以下により証明できる。

- `exec ((n-pop (suc n)) . (n-push (suc n))) m = m` を示せれば良い。
- `X` に注目した時 n-pop の定義より `exec (n-pop n) . pop . (n-push (suc n)) m = m`
- exec-comp より `exec (n-pop n) (exec pop (n-push (suc n)) m) = m`
- exec-comp より `exec (n-pop n) (exec pop (exec (n-push (suc n)) m)) = m`
- exec-comp より `exec (n-pop n) (exec pop . (n-push (suc n)) m) = m`
- pop-n-push より `exec (n-pop n) (exec (n-push n) m) = m`
- n-push-pop より `m = m` となり証明終了。
- なお、n-push-pop は (suc n) が n に減少するため、確実に停止することから
  自身を自身の証明に適用している。

第2章　まとめ

2.1　今後の課題

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

[2] 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.

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

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

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

[6] 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.

[7] 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.

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

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

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

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

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

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

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

[15] 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.

[16] 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.

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

[18] 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.

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.