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　CbC とメタ計算としての検証手法

ソフトウェアの規模が大きくなるにつれて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以外の動作をすることである。ここで期待された動作は仕様と呼ばれ、自然言語や論理によって記述される。検証とは定められた環境下においてソフトウェアが仕様を満たすことを保証することである。

ソフトウェアの検証手法にはモデル検査と定理証明がある。

モデル検査とはソフトウェアの全ての状態を数え上げ、その状態について仕様が常に真となることを確認する。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 [1] や、モデルを状態遷移系で記述する NuSMV [2]、C 言語/C++ を記号実行する CBMC [3] などが存在する。定理証明はソフトウェアが満たすべき仕様を論理式で記述し、その論理式が恒真であることを証明する。定理証明を行なうことができる言語には、依存型証明を行なう Agda [4] や Coq [5] 、 ATS2 [6] などが存在する。

モデル検査器や証明でソフトウェアを検証する際、検証を行なう言語と実装に使われる言語が異なるという問題がある。言語が異なれば二重で同じソフトウェアを記述する必要がある上、検証に用いるソースコードは状態遷移系でプログラムを記述するなど実装コードに比べて記述が困難である。検証されたコードから実行可能なコードを生成可能な検証系もあるが、既存の実装に対する検証は行なえない。そこで、当研究室では検証と実装が同一の言語で行なえる Continuation based C [7] 言語を開発している。

Continuation based C (CbC) は C 言語と似た構文を持つ言語である。CbC では処理の単位は関数ではなく CodeSegment という単位で行なわれる。CodeSegment は値を入力として受け取り出力を行なう処理単位であり、CodeSegment を接続していくことによりソフトウェアを構築していく。CodeSegment の接続処理はメタ計算として定義されており、実装や環境によって切り替えを行なうことができる。検証を行なうメタ計算を定義することにより、CodeSegment の定義を検証用に変更することなくソフトウェアの検証を行なうことができる。

本論文では CbC のメタ計算として検証手法の提案と CbC の型システムの定義を行なう。モデル検査的な検証として、状態の数え上げを行なう有限のモデル検査と仕様の定義を CbC 自身で行なう。また、証明的な検証として CbC における型システムを部分型と

して定義する。部分型を利用して CbC のプログラムが証明支援系言語 Agda 上で正しく証明可能な形で定義できることを示す。

1.1　本論文の構成

本論文ではまず第??章で Continuation based C の解説を行なう。CbC を記述するプログラミングスタイルである CodeSegment と DataSegment の解説、メタ計算と状態を数え上げるメタ計算ライブラリ akasha の解説を行なう。次に第??章で型システムについて取り上げる。型システムの定義とラムダ計算、単純型付きラムダ計算と部分型について述べる。第??章では証明支援系プログラミング言語 Agda についての解説を行なう。Agda の構文や使い方、Curry-Howard Isomorphism や Natural Deduction といった証明に関する解説も行なう。第 2 章では、部分型を用いて CbC のプログラムを Agda で記述し、証明を行なう。CodeSegment や DataSegment の Agda 上での定義や、メタ計算はどのように定義されるかを解説する。

第2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。

2.1　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 2.1 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 2.1: 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
```

2.2　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 2.2 のようになる。

リスト 2.2: 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 2.3 のようになる。

リスト 2.3: 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。

2.3　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 2.4 のようになる。

リスト 2.4: 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。

2.4　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 2.5 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 2.5: 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。

2.5 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 2.6)

リスト 2.6: 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
```

2.6　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 2.7 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 2.7: 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 2.8 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 2.8: 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 2.9 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 2.9: 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)}
```

メタの階層構造を表すと図のようになる。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。

2.7　Agda を用いた Continuation based C の検証

2.8　スタックの実装の検証

第3章　まとめ

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

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

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

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

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

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

[7] 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.

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

[9] 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.

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

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

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

[13] 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.

[14] 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.

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

[16] 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.

[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.