社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

継続を基本とする言語 CbC での Hoare Logic による健全性の考察

外間　政尊†　　河野　真治††

† 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
E-mail: †{masataka,kono}@cr.ie.u-ryukyu.ac.jp

あらまし　CbC は継続を主とする言語上で、HoareLogic をベースとした検証を提案している。OS などの手続き型の記述に対しても有効なものを目指している。Curry-Howard 対応にもとづく証明支援系である Agda を用いて検証手法の健全性を示す手法について考察する。
キーワード　プログラミング言語, CbC, Gears OS, Agda, 検証

Masataka HOKAMA† and Shinji KONO††

† Interdisciplinary Information Engineering, Graduate School of Engineering and Science, University of the Ryukyus.
†† Information Engineering, University of the Ryukyus.
E-mail: †{masataka,kono}@cr.ie.u-ryukyu.ac.jp

1. OS の検証

OS やアプリケーションの信頼性は重要である。信頼性を上げるにはプログラムが仕様を満たしていることを検証する必要がある。プログラムの検証手法として、Floyd–Hoare Logic (以下 Hoare Logic) が知られている。Hoare Logic は事前条件が成り立っているときにある関数を実行して、それが停止する際に事後条件を満たすことを確認することで、検証を行う。Hoare Logic はシンプルなアプローチであるが限定されたコマンド群や while program にしか適用されないことが多く、複雑な通常のプログラミング言語には向いていない。

当研究室では信頼性の高い言語として Continuation based C (CbC) を開発している。CbC では CodeGear、DataGear という単位を用いてプログラムを記述する。

CodeGear を Agda で継続渡しの記述を用いた関数として記述する。ここで Agda は Curry Howard 対応にもどづく定理証明系であり、それ自身が関数型プログラング言語でもある。Agda では条件を命題として記述することができるので、継続に事前条件や事後条件をもたせることができる。

既存の言語では条件は assert などで記述することになるが、その証明をそのプログラミング言語内で行うことはできない。Agda では証明そのもの、つまり命題に対する推論をλ項として記述することができるので、Hoare Logic の証明そのものを Meta CodeGear として記述できる。これは既存の言語では不可能であった。ポイントは、プログラムそのものを Agda base の CodeGear で記述できることである。CodeGear は入力と出力のみを持ち関数呼び出しせずに goto 的に継続実行する。この形式がそのまま Hoare Logic のコマンドを自然に定義する。

Hoare Logic の証明には 3 つの条件が必要である。一つは事前条件と事後条件がプログラム全体で正しく接続されていることである。ループ (ループを含む CodeGear の接続) で、事前条件と事後条件が等しく、不変条件を構成していること。さらに、ループが停止することを示す必要がある。停止しないプログラムに対しては停止性を省いた部分正当性を定義できる。

本論文では Agda 上での Hoare Logic の記述を使い、簡単な while Loop のプログラムの作成、証明を行った。この証明は停止性と証明全体の健全性を含んでいる。従来は Hoare Logic の健全性は制限されたコマンドなどに対して一般的に示すのが普通であるが、本手法では複雑な CodeGear に対して、個別の証明を Meta CodeGear として自分で記述するところに特徴がある。これにより健全性自体の証明が可能になった。

2. Continuation based C

Continuation based C [15] (以下 CbC) は CodeGear を処理の単位、DataGear をデータの単位として記述するプログラミング言語である。CbC は C 言語とほぼ同じ構文を持つが、よりアセンブラに近い記述になる。

CbC では検証しやすいプログラムの単位として DataGear と CodeGear という単位を用いるプログラミングスタイルを提

案している。

DataGear は CodeGear で扱うデータの単位であり、処理に必要なデータである。CodeGear の入力となる DataGear を Input DataGear と呼び、出力は Output DataGear と呼ぶ。

CodeGear はプログラムの処理そのもので、図 1 で示しているように任意の数の Input DataGear を参照し、処理が完了すると任意の数の Output DataGear に書き込む。

CodeGear 間の移動は継続を用いて行われる。継続は関数呼び出しとは異なり、呼び出した後に元のコードに戻らず、次の CodeGear へ継続を行う。これは、関数型プログラミングでは末尾関数呼び出しを行うことに相当する。

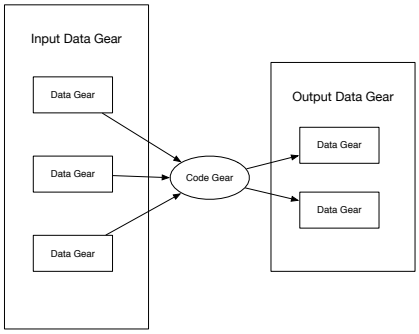

図 1: CodeGear と DataGear

また、プログラムの記述する際は、ノーマルレベルの計算の他に、メモリ管理、スレッド管理、資源管理等を記述しなければならない処理が存在する。これらの計算はノーマルレベルの計算と区別してメタ計算と呼ぶ。

メタ計算は OS の機能を通して処理することが多く、信頼性の高い記述が求められる。そのため、 CbC ではメタ計算を分離するために Meta CodeGear、 Meta DataGear を定義している。

Meta CodeGear は CbC 上でのメタ計算で、通常の CodeGear を実行する際に必要なメタ計算を分離するための単位である。図 2 のように CodeGear を実行する前後や DataGear の大枠として Meta Gear が存在している。

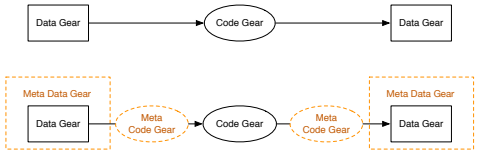

図 2: メタ計算を可視化した CodeGear と DataGear

3. 関数型言語 としての Agda

Agda [19] は純粋関数型言語である。Agda は依存型という型システムを持ち、型を第一級オブジェクトとして扱う。

Agda の記述ではインデントが意味を持ち、スペースの有無もチェックされる。コメントは `-- comment` か `{-- comment --}` のように記述される。また、_でそこに入りうるすべての値を示すことができ、?でそこに入る値や型を不明瞭なままにしておくことができる。

Agda では型をデータや関数に記述する必要がある。Agda における型指定は : を用いて `name : type` のように記述する。このとき `name` に 空白があってはいけない。データ型は、代数的なデータ構造で、その定義には `data` キーワードを用いる。`data` キーワードの後に `data` の名前と、型、 `where` 句を書きインデントを深くし、値にコンストラクタとその型を列挙する。

Code 1 は自然数の型である ℕ (Natural Number) を例である。

Code 1: 自然数を表すデータ型 `Nat` の定義

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

`Nat` では `zero` と `suc` の 2 つのコンストラクタを持つデータ型である。`suc` は ℕ を受け取って ℕ を表す再帰的なデータになっており、`suc` を連ねることで自然数全体を表現することができる。

ℕ 自身の型は `Set` であり、これは Agda が組み込みで持つ「型集合の型」である。`Set` は階層構造を持ち、型集合の集合の型を指定するには `Set1` と書く。

Agda には C 言語における構造体に相当するレコード型というデータも存在する、例えば `x` と `y` の二つの自然数からなるレコード `Point` を定義する。Code 2 のようになる。

Code 2: Agda におけるレコード型の定義

```
record Envc : Set where
  field
    vari : ℕ
    varn : ℕ
    c10  : ℕ

makeEnv : ℕ → ℕ → ℕ → Envc
makeEnv i n c = record { vari = i ; varn = n ; c10 = c }
```

レコードを構築する際は `record` キーワード後の `{}` の内部に `FieldName = value` の形で値を列挙する。複数の値を列挙するには `;` で区切る必要がある。

Agda での関数は型の定義と、関数の定義をする必要がある。関数の型はデータと同様に : を用いて `name : type` に記述するが、入力を受け取り出力返す型として記述される。→ 、または→ を用いて `input → output` のように記述される。また、`_+_`のように関数名で_を使用すると引数がその位置にあることを意味し、中間記法で関数を定義することもできる。関数の定義は型の定義より下の行に、`=` を使い `name input = output` のように記述される。

例えば引数が型 `A` で返り値が型 `B` の関数は `A → B` のように書くことができる。また、複数の引数を取る関数の型は `A → A → B` のように書ける。例として任意の自然数 ℕ を受け取り、`+1` した値を返す関数は Code 3 のように定義できる。

Code 3: Agda における関数定義

```
+1 : ℕ → ℕ
+1 m = suc m
```

```
-- eval +1 zero
-- return suc zero
```

引数は変数名で受けることもでき、具体的なコンストラクタを指定することでそのコンストラクタが渡された時の挙動を定義できる。これはパターンマッチと呼ばれ、コンストラクタで case 文を行なっているようなものである。例として自然数 ℕ の加算を関数で書くと Code 4 のようになる。

Code 4: 自然数での加算の定義

```
_+_ : ℕ → ℕ → ℕ
zero  + m = m
suc n + m = suc (n + m)
```

パターンマッチでは全てのコンストラクタのパターンを含む必要がある。例えば、自然数 ℕ を受け取る関数では `zero` と `suc` の 2 つのパターンが存在する必要がある。なお、コンストラクタをいくつか指定した後に変数で受けることもでき、その変数では指定されたもの以外を受けることができる。例えば Code 5 の減算では初めのパターンで 2 つ目の引数が `zero` のすべてのパターンが入る。

Code 5: 自然数の減算によるパターンマッチの例

```
_-_ : Nat → Nat → Nat
n     - zero = n
zero  - suc m = zero
suc n - suc m = n - m
```

Agda には λ 計算が存在している。λ 計算とは関数内で生成できる無名の関数であり、`\arg1 arg2 → function` または `λarg1 arg2 → function` のように書くことができる。Code 3 で例とした `+1` をラムダ計算で書くと Code 6 の `$\lambda$+1` ように書くことができる。この二つの関数は同一の動作をする。

Code 6: Agda におけるラムダ計算

```
+1 : ℕ → ℕ
+1 n = suc n -- not use lambda

λ+1 : ℕ → ℕ
λ+1 = (\n → suc n)   -- use lambda
```

Agda では特定の関数内のみで利用できる関数を `where` 句で記述できる。スコープは `where` 句が存在する関数内部のみであるため、名前空間が汚染させることも無い。例えば自然数 3 つを取ってそれぞれ 3 倍して加算する関数 `f` を定義するとき、`where` を使うとリスト Code 7 のように書ける。これは `f'` と同様の動作をする。`where` 句は利用したい関数の末尾にインデント付きで `where` キーワードを記述し、改行の後インデントをして関数内部で利用する関数を定義する。

Code 7: Agda における where 句

```
f : Int → Int → Int
f a b c = (t a) + (t b) + (t c)
  where
    t x = x + x + x

f' : Int → Int → Int
f' a b c = (a + a + a) + (b + b + b) + (c + c + c)
```

また Agda では停止性の検出機能が存在し、プログラム中に停止しない記述が存在するとコンパイル時にエラーが出る。`{-# TERMINATING #-}`のタグを付けると停止しないプログラムをコンパイルすることができるがあまり望ましくない。Code 8 で書かれた、`loop` と `stop` は任意の自然数を受け取り、0 になるまでループして 0 を返す関数である。`loop` では ℕ の数を受け取り、 `loop` 自身を呼び出しながら 数を減らす関数 pred を呼んでいる。しかし、`loop` の記述では関数が停止すると言えないため、定義するには`{-# TERMINATING #-}`のタグが必要である。`stop` では自然数がパターンマッチで分けられ、`zero` のときは `zero` を返し、 `suc n` のときは `suc` を外した `n` で stop を実行するため停止する。

Code 8: 停止しない関数 loop、停止する関数 stop

```
{-# TERMINATING #-}
loop : ℕ → ℕ
loop n = loop (pred n)

-- pred : ℕ → ℕ
-- pred zero    = zero
-- pred (suc n) = n

stop : ℕ → ℕ
stop zero = zero
stop (suc n) = (stop n)
```

このように再帰的な定義の関数が停止するときは、何らかの値が減少する必要がある。

4. 定理証明支援器としての Agda

Agda での証明では関数の記述と同様の形で型部分に証明すべき論理式、λ 項部分にそれを満たす証明を書くことで証明を行うことが可能である。証明の例として Code Code 9 を見る。ここでの `+zero` は右から `zero` を足しても ≡ の両辺は等しいことを証明している。これは、引数として受けている `y` が `Nat` なので、 `zero` の時と `suc y` の二つの場合を証明する必要がある。

Code 9: 等式変形の例

```
+zero : { y : ℕ } → y + zero  ≡ y
+zero {zero} = refl
+zero {suc y} = cong suc ( +zero {y} )
```

`y = zero` の時は $zero \equiv zero$ とできて、左右の項が等しいということを表す `refl` で証明することができる。`y = suc y` の時は $x \equiv y$ の時 $fx \equiv fy$ が成り立つという Code 10 の `cong` を使って、y の値を 1 減らしたのち、再帰的に `+zero y` を用いて証明している。

Code 10: cong

```
cong : ∀ (f : A → B) {x y} → x ≡ y → f x ≡ f y
cong f refl = refl
```

また、他にも λ 項部分で等式を変形する構文がいくつか存在

している。ここでは `rewrite` と `≡-Reasoning` の構文を説明するとともに、等式を変形する構文の例として加算の交換則について示す。

`rewrite` では 関数の `=` 前に `rewrite` 変形規則 の形で記述し、複数の規則を使う場合は `rewrite` 変形規則 1 | 変形規則 2 のように |を用いて記述する。Code `??` にある `+-comm` で `x` が `zero` のパターンが良い例である。ここでは、`+zero` を利用し、 `zero + y` を `y` に変形することで $y \equiv y$ となり、左右の項が等しいことを示す `refl` になっている。

Code 11:　等式変形の例 3/3

```
+−comm : (x y : ℕ) → x + y ≡ y + x
+−comm zero y rewrite (+zero {y}) = refl
+−comm (suc x) y = let open ≡−Reasoning in
  begin
  suc (x + y) ≡⟨⟩
  suc (x + y) ≡⟨ cong suc (+−comm x y) ⟩
  suc (y + x) ≡⟨ sym (+−suc {y} {x}) ⟩
  y + suc x ■

−− +−suc : {x y : ℕ} → x + suc y ≡ suc (x + y)
−− +−suc {zero} {y} = refl
−− +−suc {suc x} {y} = cong suc (+−suc {x} {y})
```

Code 11 では `suc (y + x)` *equiv* `y + (suc x)` という等式に対して *equiv* の対称律 `sym` を使って左右の項を反転させ `y + (suc x)` *equiv* `suc (y + x)` の形にし、`y + (suc x)` が `suc (y + x)` に変形できることを `+-suc` を用いて示した。これにより等式の左右の項が等しくなったため `+-comm` が示せた。

Agda ではこのような形で等式を変形しながら証明を行う事ができる。

5. Hoare Logic

Floyd-Hoare Logic [14](以下 Hoare Logic) とは C.A.R Hoare、 R.W Floyd が考案したプログラムの検証の手法である。

Hoare Logic では事前条件が成り立つとき、何らかの計算 (以下コマンド) を実行した後に事後条件が成り立つことを検証する。事前条件を P 、 何らかの計算を C 、 事後条件を Q としたとき、

$$\{P\}\ C\ \{Q\}$$

といった形で表される。この三組は Hoare Triple と呼ばれる。

Code 5. は while program の例である。これは変数 `n` と `i` を持ち、`n` が 0　より大きいとき、`i` を増やし `n` を減らす、疑似プログラムである。

```
n = 10;
i = 0;

while (n > 0) {
  i++;
  n−−;
}
```

このプログラムでの状態は、初めの $n = 10$、 $i = 0$ を代入する条件、while loop 中に成り立っている条件を $n + i = 10$、while loop が終了したとき成り立っている条件を $i = 10$ としている。

CbC 上での Hoare Logic で同様のプログラムを作成し、検証を行う。

6. DataGear、 CodeGear と Agda の対応

現在 CbC では検証用の上位言語として Agda を利用しており、Agda では CbC のプログラムをメタ計算を含む形で記述することができる。

Agda での DataGear は Agda で使うことのできるすべてのデータに対応する。また、Agda での記述はメタ計算として扱われる。

CodeGear は DataGear を受け取って処理を行い DataGear を返す。また、CodeGear 間の移動は継続を用いて行われる。継続は関数呼び出しとは異なり、呼び出した後に元のコードに戻らず、次の CodeGear へ継続を行うものであった。

これは、関数型プログラミングでは末尾関数呼び出しを行うことに相当し、継続渡し (Continuation Passing Style) で書かれた Agda の関数と対応する。継続は不定の型 (`t`) を返す関数で表される。継続先は次に実行する関数の型を引数として受け取り不定の型 `t` を返す関数として記述され、CodeGear 自体も同じ型 `t` を返す関数となる。

Code 12 は Agda で記述した加算を行う CodeGear の例である。

Code 12:　`Agda` での `CodeGear` の例

```
plus : {l : Level} {t : Set l} → (x y : ℕ) → (next : ℕ → t)
      → t
plus x zero next = next x
plus x (suc y) next = plus (suc x) y next
```

`plus 10 20` を評価すると `next` に 30 が入力されていることがわかる。

7. CbC での Hoare Logic の記述

Hoare Logic の事前条件や事後条件は変数の大小関係や同値関係などで表される。Agda 上では関係もデータとして扱うことができるため、関係を引数とした CodeGear を用いてプログラムを記述する。

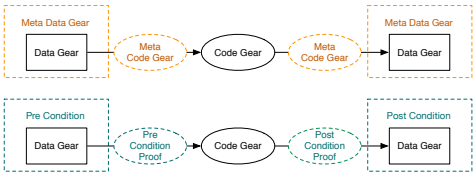

図 3: CodeGear、DataGear での Hoare Logic

CbC での Hoare Logic は fig 3 が示すように、事前条件 (Pre Condition) が Proof で成立しており、 CodeGear で変更し、事後条件 (Post Condition) が成り立つことを Proof で検証している。

13 は通常の CodeGear と Hoare Logic ベースの CodeGear

を例としている。通常の CodeGear である `whileLoop'` と Hoare Logic ベースの CodeGear である `whileLoopPwP'` は同じ動作をする。

Code 13: CbC 上での Hoare Logic

```
-- Nomal CodeGear
whileLoop' : {l : Level} {t : Set l} → (n : ℕ) → (env : Envc
    )
  → (n ≡ varn env)
  → (next : Envc → t)
  → (exit : Envc → t) → t
whileLoop' zero env refl _ exit = exit env
whileLoop' (suc n) env refl next _ = next (record env {
      varn = pred (varn env) ; vari = suc (vari env) })

-- Hoare Logic base CodeGear
whileLoopPwP' : {l : Level} {t : Set l} → (n : ℕ) → (env :
    Envc )
  → (n ≡ varn env) → (pre : varn env + vari env ≡ c10
    env)
  → (next : (env : Envc ) → (pred n ≡ varn env) → (post :
     varn env + vari env ≡ c10 env) → t)
  → (exit : (env : Envc ) → (fin : vari env ≡ c10 env)  → t)
     → t
whileLoopPwP' zero env refl refl next exit = exit env refl
whileLoopPwP' (suc n) env refl refl next exit = next (
    record env {varn = pred (varn env) ; vari = suc (vari
    env) }) refl (+−suc n (vari env))
```

`whileLoopPwP'` では引数として事前条件 pre と継続先の関数を受け取っており、継続先の関数が受け取る引数 post や fin などの条件がこの関数においての事後条件となる。

8. CbC 上での Hoare Logic を用いた仕様記述と停止性

Hoare Logic では用意されたシンプルなコマンドを用いてプログラムを記述したが、CbC 上では CodeGear という単位でプログラムを記述する。そのため Hoare Logic のコマンドと同様に CodeGear を使った仕様記述を行う必要がある。

while program には初めの $n = 10$、$i = 0$ を代入する条件、while loop 中に成り立っている条件を $n + i = 10$、while loop が終了したとき成り立っている条件を $i = 10$ の 3 つの状態があった。

Code 14 は while program の 3 つの状態を記述したものである。

Code 14: CbC ベースの Hoare Logic

```
data whileTestState  : Set where
  s1 : whileTestState
  s2 : whileTestState
  sf : whileTestState

whileTestStateP : whileTestState → Envc →  Set
whileTestStateP s1 env = (vari env ≡ 0) ∧ (varn env ≡ c10
    env)
whileTestStateP s2 env = (varn env + vari env ≡ c10 env)
whileTestStateP sf env = (vari env ≡ c10 env)
```

`whileTestStateP` では `s1` が初期状態、`s2` がループ内不変条件、`fs` が最終状態に対応している。`s1`、`s2`、`s3` はそれぞれ `whileTestState` で定義された識別子である。

これらの状態を使って、CbC 上の Hoare Logic を使って while program を作成していく。

Code 15 は代入部分の Meta CodeGear である。代入では事前条件がなく、事後条件として `s1` の `(vari env ≡ 0) ∧ (varn env ≡ c10 env)` が成り立つ。

Code 15: CbC 上の Hoare Logic での 代入

```
whileTestPwP : {l : Level} {t : Set l} → (c10 : ℕ) → ((env
    : Envc ) → whileTestStateP s1 env → t) → t
whileTestPwP c10 next = next env record { pi1 = refl ; pi2
    = refl } where
  env : Envc
  env = whileTestP c10 ( λ env → env )
```

Code 16 はループを行うコードである。`whileLoopP'` はループを続ける、終えるの判断を行う Meta CodeGear で、ループを続けている間、`varn` の値を減らし、`vari` の値を増やしている。ループは `varn` が `suc n` の間続き、その間の条件である `s2`、つまり `(varn env + vari env ≡ c10 env)` の状態が成り立つ。`varn` が `zero` になると最後の `loopPwP'` に `fs` である `(vari env ≡ c10 env)` を渡し、ループを終える。

`loopPwP'` は実際にループをする Meta CodeGear で、回って来た際に `varn` が `suc n` の間は `whileLoopPwP'` を実行し、その継続先の Meta CodeGear に自身である `loopPwP'` を入れてループを行う。`varn zero` のケースはその前の `whileLoopPwP'` が `zero` で `sf` の最終状態を返してくるため、`loopPwP'` でも同様に `sf` である `(vari env ≡ c10 env)` を返し、ループが終了する。

Code 16: CbC 上の Hoare Logic での while loop

```
whileLoopPwP' : {l : Level} {t : Set l} → (n : ℕ) → (env :
    Envc )
   → (n ≡ varn env) → whileTestStateP s2 env
   → (next : (env : Envc ) → (pred n ≡ varn env) →
     whileTestStateP s2 env  → t)
   → (exit : (env : Envc ) → whileTestStateP sf env  → t)
     → t
whileLoopPwP' zero env refl refl _ exit = exit env refl
whileLoopPwP' (suc n) env refl refl next _ =
   next (record env {varn = pred (varn env) ; vari = suc (
     vari env) }) refl (+−suc n (vari env))


loopPwP' : {l : Level} {t : Set l} → (n : ℕ) → (env : Envc
    )
   → (n ≡ varn env) → whileTestStateP s2 env
   → (exit : (env : Envc ) → whileTestStateP sf env → t)
     → t
loopPwP' zero env refl refl exit = exit env refl
loopPwP' (suc n) env refl refl exit
   = whileLoopPwP' (suc n) env refl refl (λ env x y →
     loopPwP' n env x y exit) exit
```

これらの Meta CodeGear を使い仕様を記述する。

Code 17: CbC 上の Hoare Logic

```
whileTestPCallwP' : (c :  ℕ ) → Set
whileTestPCallwP' c = whileTestPwP {_} {_} c (λ env s
    → loopPwP' (varn env) env refl (conv env s) ( λ env s
     → vari env ≡ c10 env )  )
```

```
-- conv : (env : Envc ) → (vari env ≡ 0) ∧ (varn env ≡
      c10 env) → varn env + vari env ≡ c10 env
-- conv e record { pi1 = refl ; pi2 = refl } = +zero
```

`whileTestPCallwP'` は Code 15 や Code 16 で解説した Meta CodeGear を組み合わせた仕様である。

また、while program と同様にループ内ではそのままの条件だとループさせることが難しいため `conv` を使ってループ内不変条件へと変化させている。

この仕様では `whileTestPwP` と `loopPwP'` を続けて実行したとき、最後のラムダ式に入っている最終状態 `vari env ≡ c10 env` が必ず成り立つ。

`loopHelper` では `loopPwP'` が必ず停止し、`vari env ≡ c10 env` が成り立つことが分かる。

Code 18: `loopHelper` を使った停止性

```
whileCallwP : (c : ℕ) → whileTestPCallwP' c
whileCallwP c = whileTestPwP {_} {_} c
  (λ env s → loopHelper c (record { c10 = c ; varn = c ;
      vari = zero }) refl +zero)
```

9. CbC 上での Hoare Logic を用いた Soundness の証明

ここでは CbC 上の Hoare Logic での Soundness(健全性) について確認する。

Code 19 は A ならば B の命題により証明を行う implies である。implies は Set である A と B を受け取る。このとき A → B が存在すれば A implies B を証明することができる。

Code 19: `implies`

```
data _implies_  (A B : Set ) : Set (succ Zero) where
  proof : ( A → B ) → A implies B
```

代入を行う CodeGear である whileTestP は初期状態を持たないため、常に真の命題 ⊤ と代入を終えたときの事後条件である (vari env ≡ 0) ∧ (varn env ≡ c10 env) を implies に入れた型を記述する。Code20 は実際に implies を用いて記述した証明である。証明では proof に (⊤ → (vari env ≡ 0) ∧ (varn env ≡ c10 env)) であると記述できればよく、

ここでは λ _ → record pi1 = refl ; pi2 = refl がこれに対応する。

Code 20: 代入の `implies` による証明

```
whileTestPSem :  (c : ℕ) → whileTestP c
  ( λ env → ⊤ implies (vari env ≡ 0) ∧ (varn env ≡ c10
      env) )
whileTestPSem c = proof ( λ _ → record { pi1 = refl ; pi2
      = refl } )
```

Code21 の whileTestPSemSound は output ≡ whileTestP c (λ e → e) を受け取ることで whileTestP の実行が終わった結果、つまり停止した CodeGear の実行結果が事後条件を満たしていることを証明している

Code 21: 停止性を考慮した代入の健全性の証明

```
whileTestPSemSound : (c : ℕ ) (output : Envc ) →
  output ≡ whileTestP c (λ e → e)
  → ⊤ implies ((vari output ≡ 0) ∧ (varn output ≡ c))
whileTestPSemSound c output refl = whileTestPSem c
```

同様に他の CodeGear に対しても健全性の証明が可能である。Code 22 の whileConvPSemSound は制約を緩める conversion の健全性の証明である。

Code 22: `conversion` の `implies` による証明

```
whileConvPSemSound : {l : Level} → (input : Envc) → ((
      vari input ≡ 0) ∧ (varn input ≡ c)) implies (varn
      input + vari input ≡ c10 input)
whileConvPSemSound input = proof λ x → (conversion
      input x) where
  conversion : (env : Envc ) → (vari env ≡ 0) ∧ (varn env
      ≡ c10 env) → varn env + vari env ≡ c10 env
  conversion e record { pi1 = refl ; pi2 = refl } = +zero
```

conversion でも同様に元のプログラムに対して証明を行えている。

Code 23 は while loop でのループをする CodeGear である。ここでは loopPPSemInduct という補助定理を使って証明を記述しているが、この証明は長くなったため、元のソースコード [9] を参照していただきたい。

Code 23: `loop` の `implies` による証明

```
loopPPSem : (input output : Envc ) →  output ≡ loopPP (
      varn input)  input refl
  → (varn input + vari input ≡ c10 input )
  → (varn input + vari input ≡ c10 input ) implies (vari
      output ≡ c10 output)
loopPPSem input output refl s2p = loopPPSemInduct (
      varn input) input  refl refl s2p
```

24 はループの判断をする CodeGear で loopPPSem を使って証明を行っている。

Code 24: `loop` の `implies` による停止性を含めた証明

```
whileLoopPSemSound : {l : Level} → (input output : Envc
      )
  → (varn input + vari input ≡ c10 input)
  →  output ≡ loopPP (varn input) input refl
  → (varn input + vari input ≡ c10 input) implies (vari
      output ≡ c10 output)
whileLoopPSemSound {l} input output pre eq =
      loopPPSem input output eq pre
```

ここでも同様に元のプログラムに対して証明を行うことができた。

CbC 上での Hoare Logic では implies を用いて健全性に対する証明が行えると考えている。

10. まとめと今後の課題

本論文では Continuation based C プログラムに対して Hoare Logic をベースにした仕様記述と検証を行った。また、CbC での Hoare Logic では仕様を含めた記述のまま、実際にコードが実行できることを確認した。

実際に、Hoare Logic ベースの記述を行うことで、検証のメタ計算に使われる Meta DataGear や、CodeGear の概念が明確となった。また、CbC 上での Pre Condition、Post Condition の記述方法が明確になった。

元の Hoare Logic ではコマンドのみでのプログラム記述と検証を行っていたが、CodeGear をベースにすることでより柔軟な単位でのプログラム記述し、実際に検証を行えることが分かった。

以前は検証時に無限ループでなくてもループが存在すると、Agda が導出時に step での実行を行うため、ループ回数分 step を実行する必要があったが、ループに対する簡約を記述することで、有限回のループを抜けて証明が記述できることが判明した。今後、ループ構造に対する証明は同様に解決できると考えられるため、より多くの証明が可能となると期待している。

今後の課題として、他のループが発生するプログラムの検証が挙げられる。同様に検証が行えるのであれば、共通で使えるライブラリのような形でまとめることで、より容易な検証ができるようになるのではないかと考えている。現在、検証が行われていないループが存在するプログラムとして、Binary Tree や RedBlack Tree などのデータ構造が存在するため、それらのループに対して今回の手法を適用して検証を行いたい。

また、Meta DataGear で DataGear の関係等の制約条件を扱うことで、常に制約を満たすデータを作成することができる。予めそのようなデータをプログラムを使用することで、検証を行う際の記述減らすことができると考えている。これも同様に Binary Tree や RedBlack Tree などのデータ構造に適用し、検証の一助になると考えている。

その他の課題としては、CbC で開発されている GearsOS に存在する並列構文の検証や、検証を行った Agda 上の CbC 記述からノーマルレベルの CbC プログラムの生成などが挙げられる。

文　献

[1] The agda wiki. http://wiki.portal.chalmers.se/agda/pmwiki.php. Accessed: 2018/12/17(Mon).

[2] Agda1. https://sourceforge.net/projects/agda/. Accessed: 2020/2/9(Sun).

[3] Ats-pl-sys. http://www.ats-lang.org/. Accessed: 2020/2/9(Sun).

[4] cbc-gcc - 並列信頼研 mercurial repository. http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_gcc/. Accessed: 2020/2/9(Sun).

[5] cbc-llvm - 並列信頼研 mercurial repository. http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_llvm/. Accessed: 2020/2/9(Sun).

[6] Coq source. https://github.com/coq/coq. Accessed: 2020/2/9(Sun).

[7] Example - hoare logic. http://ocvs.cfv.jp/Agda/readmehoare.html. Accessed: 2019/1/16(Wed).

[8] Hoare logic in agda2. https://github.com/IKEGAMIDaisuke/HoareLogic. Accessed: 2020/2/9(Sun).

[9] loopseminduct - 並列信頼研 mercurial repository. http://www.cr.ie.u-ryukyu.ac.jp/hg/Members/ryokka/HoareLogic/file/tip/whileTestGears.agda. Accessed: 2020/2/9(Sun).

[10] Rust programming language. https://www.rust-lang.org/. Accessed: 2020/2/9(Sun).

[11] Welcome! | the coq proof assistant. https://coq.inria.fr/. Accessed: 2020/2/9(Sun).

[12] Welcome to agda's documentation! — agda latest documentation. http://agda.readthedocs.io/en/latest/. Accessed: 2018/12/17(Mon).

[13] whiletestprim.agda - 並列信頼研 mercurial repository. http://www.cr.ie.u-ryukyu.ac.jp/hg/Members/ryokka/HoareLogic/file/tip/whileTestPrim.agda. Accessed: 2020/2/9(Sun).

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[15] Tokumori Kaito and Kono Shinji. Implementing continuation based language in llvm and clang. *LOLA 2015, Kyoto*, July 2015.

[16] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, June 2010.

[17] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.

[18] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 252–269, New York, NY, USA, 2017. ACM.

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

[20] Aaron Stump. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2016.

[21] 伊波立樹. Gears os の並列処理. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2018.

[22] 政尊 外間 and 真治 河野. Gearsos の agda による記述と検証. Technical Report 5, 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, may 2018.

[23] 宮城光希. 継続を基本とした言語による os のモジュール化. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2019.

[24] 宮城光希 and 河野真治. Code gear と data gear を持つ gears os の設計. In **第 *59* 回プログラミング・シンポジウム予稿集**, volume 2018, pages 197–206, jan 2018.

[25] 比嘉 健太 and 河野 真治. Verification method of programs using continuation based c. **情報処理学会論文誌プログラミング（*PRO*）**, 10(2):5–5, feb 2017.

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

[27] 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.

[28] 比嘉健太. メタ計算を用いた continuation based c の検証手法. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2017.