修士 (工学) 学位論文

**Master's Thesis of Engineering**

# Continuation based C での HoareLogic を用いた仕様記述と検証

2020 年 3 月

March 2020

外間 政尊

**Masataka HOKAMA**

琉球大学

大学院理工学研究科

情報工学専攻

**Information Engineering Course**

**Graduate School of Engineering and Science**

**University of the Ryukyus**

指導教員 : 教授 玉城 史朗

**Supervisor: Prof. Shirou TAMAKI**

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)　　玉城　史朗

印

(副　査)　　山田　孝治

印

(副　査)　　當間　愛晃

印

(副　査)　　河野　真治

# 要旨

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) が存在している。HoareLogic 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。しかし、 HoareLogic はシンプルなアプローチであるが通常のプログラミング言語で使用することができず、広まっているとはいえない。

当研究室では信頼性の高い OS として GearsOS を開発している。現在 GearsOS では CodeGear、DataGear という単位を用いてプログラムを記述する手法を用いており、仕様の確認には定理証明系である Agda を用いている。

CodeGear は Agda 上では継続渡しの記述を用いた関数として記述する。また、継続にある関数を実行するための事前条件や事後条件などをもたせることが可能である。

そのため Hoare Logic と CodeGear、DataGear という単位を用いたプログラミング手法記述とは相性が良く、既存の言語とは異なり HoareLogic を使ったプログラミングが容易に行えると考えている。

本研究では Agda 上での HoareLogic 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。また、GearsOS の仕様確認のために CodeGear、DataGear という単位を用いた記述で Hoare Logic をベースとした while Loop プログラムを記述、その証明を行なった。

# 研究関連業績

1. 外間政尊, 河野真治. GearsOS の Agda による記述と検証. 研究報告システムソフトウェアとオペレーティング・システム（OS), May, 2018
2. 外間政尊, 河野真治. GearsOS の Hoare Logic をベースにした検証手法. 電子情報通信学会 ソフトウェアサイエンス研究会 (SIGSS) 1 月, Jan, 2019
3. 外間政尊, 河野真治. 継続を基本とする言語 CbC での HoareLogic による健全性の考察. 電子情報通信学会 ソフトウェアサイエンス研究会 (SIGSS) 3 月, Mar, 2020

# 目 次

# 図 目 次

# 表 目 次

# ソースコード目次

# 第1章　プログラミング言語の検証

現在の OS やアプリケーションの検証では、実装と別に検証用の言語で記述された実装と証明を持つのが一般的である。kernel 検証 [1],[2] の例では C で記述された Kernel に対して、検証用の別の言語で書かれた等価な kernel を用いて OS の検証を行っている。また、別のアプローチとしては ATS2[3] や Rust[4] などの低レベル記述向けの関数型言語を実装に用いる手法が存在している。

証明支援向けのプログラミング言語としては Agda[5]、 Coq[6] などが存在しているが、これらの言語自体は実行速度が期待できるものではない。

そこで、当研究室では検証と実装が同一の言語で行う Continuation based C[7] という言語を開発している。Continuation based C(CbC) では、処理の単位を CodeGear、データの単位を DataGear としている。CodeGear は値を入力として受け取り出力を行う処理の単位であり、 CodeGear の出力を 次の GodeGear に接続してプログラミングを行う。CodeGear 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うことができる。このメタ計算部分で assertion などの検証を行うことで、 CodeGear の処理に手を加えることなく検証を行う。現段階では CbC 自体に証明を行うためのシステムが存在しないため、証明支援系言語である Agda を用いて等価な実装の検証を行っている。

# 第2章　Continuation based C

Continuation based C[7] (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 のプログラミングでは DataGear を CodeGear で変更し、その変更を次の CodeGear に渡して処理を行う。現在 CbC の処理系には llvm/clang による実装 [8] と gcc [9] による実装が存在する。

本章は CbC の概要についての説明する。

## 2.1　Code Gear と Data Gear

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 2.1 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。

## 2.2　Meta CodeGear、 Meta DataGear

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。

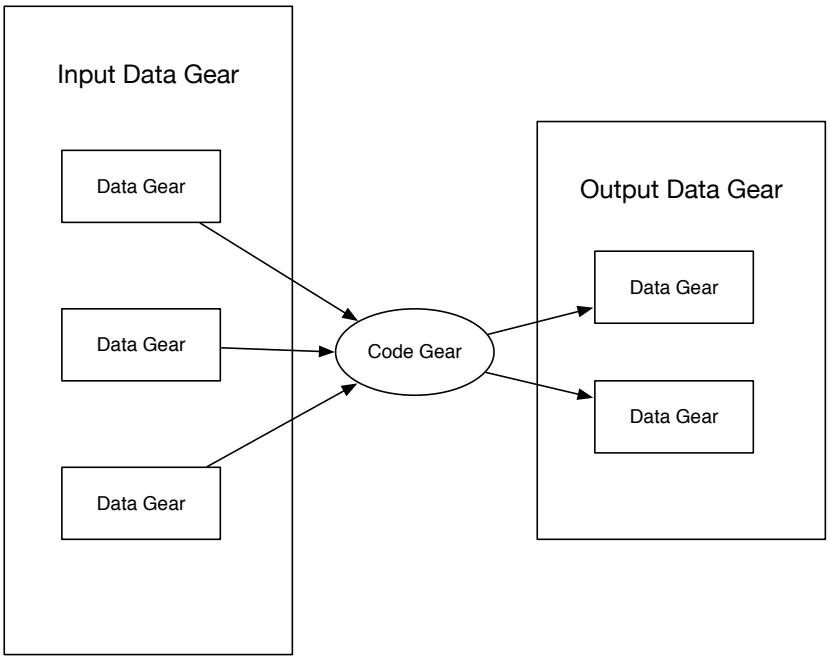


図 2.1: CodeGear と DataGear

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 2.2 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。

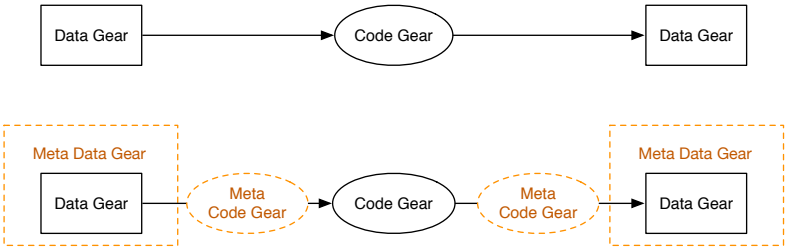


図 2.2: メタ計算を可視化した CodeGear と DataGear

例として CodeGear が DataGear から値を取得する際に使われる Meta CodeGear である stub CodeGear について説明する。CbC では CodeGear を実行する際、ノーマルレベルの計算からは見えないが必要な DataGear を Context と呼ばれる Meta DataGear を通して取得することになる。これはユーザーが直接データを扱える状態では信頼性が

高いとは言えないと考えるからである。そのために、 Meta CodeGear を用いて Context から必要な DataGear を取り出し、 CodeGear に接続する stub CodeGear という Meta CodeGear が定義されている。

Meta DataGear は CbC 上のメタ計算で扱われる DataGear である。例えば stub CodeGear では Context と呼ばれる接続可能な CodeGear、DataGear のリストや、DataGear のメモリ空間等を持った Meta DataGear を扱っている。

# 第3章　定理証明支援系言語 Agda

Agda [5] とは定理証明支援器であり、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ことが可能である。また、型システムは Curry-Howard 同型対応により命題と型付きラムダ計算が一対一で対応するため Agda では記述したプログラムを証明することができる。

本章では Agda で証明をするために必要な要素を示し。また、Agda での証明について説明する。

## 3.1　関数型言語としての Agda

Agda [5] 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 のプログラムは全てモジュール内部に記述される。そのため、各ファイルのトップレベルにモジュールを定義する必要がある。トップレベルのモジュールはファイル名と同一になる。

モジュール内で異なるモジュールをインポートする時は `import` キーワードを指定する。インポートを行なう際、モジュール内部の関数を別名に変更するには `as` キーワードを用いる。他にも、モジュールから特定の関数のみをインポートする場合は `using` キーワード、関数名を、関数の名前を変える時は `renaming` キーワードを、特定の関数のみを隠す場合は `hiding` キーワードを用いる。なお、モジュールに存在する関数をトップレベルで用いる場合は `open import` キーワードを使うことで展開できる。モジュールをインポートする例をソースコード 3.1 に示す。

ソースコード 3.1: モジュールのインポートとオプション

```
1 import Data.Nat                    -- import module
2 import Data.Bool as B              -- renamed module
3 import Data.List using (head)      -- import Data.head function
```

```
4 import Level renaming (suc to S) -- import module with rename suc to S
5 import Data.String hiding (_++_) -- import module without _++_
6 open import Data.List            -- import and expand Data.List
```

## 3.2　Agda のデータ

Agda 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。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。

ソースコード 3.2 は自然数の型である ℕ (Natural Number) を例である。

ソースコード 3.2: 自然数を表すデータ型 Nat の定義

```
1 data ℕ : Set where
2   zero : ℕ
3   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。ソースコード 3.3 のようになる。

ソースコード 3.3: Agda におけるレコード型の定義

```
1 record Point : Set where
2   field
3     x : Nat
4     y : Nat
5
6 makePoint : Nat → Nat → Point
7 makePoint a b = record { x = a ; y = b }
```

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。

## 3.3　Agda の関数

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。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。この時の型は `A → (A → B)` のように考えられる。例として任意の自然数 $mathbbN$ を受け取り、`+1` した値を返す関数はソースコード 3.4 のように定義できる。

ソースコード 3.4: Agda における関数定義

```
1 +1 : ℕ → ℕ
2 +1 m = suc m
3
4 -- eval +1 zero
5 -- 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 $mathbbN$ の加算を関数で書くとソースコード **??**のようになる。

ソースコード 3.5: 自然数での加算の定義

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

`_+_`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する。

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 $mathbbN$ 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。例えばソースコード 3.6 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。

ソースコード 3.6: 自然数の減算によるパターンマッチの例

```
1 _-_ : Nat → Nat → Nat
2 n     - zero = n      -- "n - zero" have 2-pattern, "zero - zero" "suc n
     - zero"
3 zero  - suc m = zero
4 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` のように書くことができる。ソースコード **??** で例とした `+1`

をラムダ計算で書くとソースコード 3.7 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。

ソースコード 3.7: Agda におけるラムダ計算

```
1 +1 : ℕ → ℕ
2 +1 n = suc n -- not use lambda
3
4 λ+1 : ℕ → ℕ
5 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 3.8 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。

ソースコード 3.8: Agda における where 句

```
1 f : Int → Int → Int
2 f a b c = (t a) + (t b) + (t c)
3   where
4     t x = x + x + x
5
6 f' : Int → Int → Int
7 f' a b c = (a + a + a) + (b + b + b) + (c + c + c)
```

## 3.4　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 ソースコード 3.9 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。

y = zero の時は両辺が 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 cong を使って、y の値を 1 減らしたのちに再帰的に +zero y を用いて証明している。

ソースコード 3.9: 等式変形の例

```
1 +zero : { y : ℕ } → y + zero  ≡ y
2 +zero {zero} = refl
3 +zero {suc y} = cong ( λ x → suc x ) ( +zero {y} )
```

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。ソースコード 3.10 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。

ソースコード 3.10: rewrite での等式変形の例

```
1 rewrite-+-comm : (x y : ℕ) → x + y ≡ y + x
2 rewrite-+-comm zero y rewrite (+zero {y}) = refl
3 rewrite-+-comm (suc x) y = ?
```

ソースコード 3.11、ソースコード 3.12、ソースコード 3.13 は≡-Reasoning を用いた等式変形の流れである。始めに等式変形を始めたいところで let open ≡ -Reasoning in begin と記述し、変形前 ≡⟨ 変形規則 ⟩ 変形後 の形で記述して、最後に ■ をつけて変形を終える。この let open から ■ までの流れは 1 行で記述しても良いし、改行やインデントを含めても良い。ソースコード 3.11 の例では分からないところを ? と置いておき、? の中で示されている値は下にコメントで示しておく。

ソースコード 3.11: 等式変形の例 1/3

```
1  +-comm : (x y : ℕ) → x + y ≡ y + x
2  +-comm zero y rewrite (+zero {y}) = refl
3  +-comm (suc x) y = let open ≡-Reasoning in
4    begin
5    ?0 ≡⟨ ?1 ⟩
6    ?2 ■
7
8  -- ?0 : ℕ {(suc x) + y}
9  -- ?1 : suc x + y ≡ y + suc x
10 -- ?2 : ℕ
```

この状態で実行すると ? 部分に入る型を Agda が示してくれる。始めに変形する等式を ?0 に記述し、?1 の中に変形規則を使用することで等式を変形できる。ここでの方針は (suc x) + y を suc (x + y) 変形してやり、y + (suc x) も同様に suc (x + y) の形に変形することで等しさを証明する。Agda の加算では左側に suc がついていた場合外に suc を出して再帰的に中身と足し算を行うため、何もせずに (suc x) + y は suc (x + y) に変換できる。ソースコード 3.12 では suc (x + y) に対して cong で suc を外に出し +comm を再帰的に利用することで suc (y + x) へ変換している。

ソースコード 3.12: 等式変形の例 2/3

```
1 +-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) ≡⟨ ?0 ⟩
  ?1 ■

-- ?0 : suc (y + x) ≡ y + suc x
-- ?1 : y + suc x
```

ソースコード 3.13 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。

ソースコード 3.13: 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})
```

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。

# 第4章　Hoare Logic

Floyd-Hoare Logic [10](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、*PCQ* といった形で表される。Hoare Logic ではプログラムの部分的な正当性を検証することができ、事後条件のあとに別の Command をつなげてプログラムを構築することで、シンプルな計算に対する検証することができる。

本章は Agda で実装された Hoare Logic について解説し、実際に Hoare Logic を用いた検証を行う。

## 4.1　Hoare Logic

現在 Agda 上での Hoare Logic は初期の Agda で実装されたもの [11] とそれを現在の Agda に対応させたもの [12] が存在している。

例として現在 Agda に対応させたもの [12] の Command と証明のためのルールを使って Hoare Logic を実装した。4.2 は Agda 上での Hoare Logic の構築子である。

例として Code 4.1 のようなプログラムを記述した。

ソースコード 4.1: while Loop Program

```
1  n = 10;
2  i = 0;
3
4  while (n>0)
5  {
6    i++;
7    n--;
8  }
```

*Env* は Code 4.1 の n、 i といった変数をまとめたものであり、型として Agda 上での自然数の型である Nat を持つ。

PrimComm は Primitive Command で、 n、i といった変数に 代入するときに使用される関数である。

Cond は Hoare Logic の Condition で、 Env を受け取って　Bool 値を返す関数となっている。

Agda のデータで定義されている Comm は Hoare Logic での Command を表す。

Skip は何も変更しない Command で、 Abort はプログラムを中断する Command である。

PComm は PrimComm を受けて Command を返す型で定義されており、 変数を代入するときに使われる。

Seq は Sequence で Command を 2 つ受けて Command を返す型で定義されている。これは、ある Command から Command に移り、その結果を次の Command に渡す型になっている。

If は Cond と Comm を 2 つ受け取り、 Cond が true か false かで 実行する Comm を変える Command である。

While は Cond と Comm を受け取り、 Cond の中身が True である間、 Comm を繰り返す Command である。

ソースコード 4.2: Agda での Hoare Logic の構成

```
1  PrimComm : Set
2  PrimComm = Env → Env
3
4  Cond : Set
5  Cond = (Env → Bool)
6
7  data Comm : Set where
8    Skip  : Comm
9    Abort : Comm
10   PComm : PrimComm -> Comm
11   Seq   : Comm -> Comm -> Comm
12   If    : Cond -> Comm -> Comm -> Comm
13   While : Cond -> Comm -> Comm
```

Agda 上の Hoare Logic で使われるプログラムは Comm 型の関数となる。プログラムの処理を Seq でつないでいき、最終的な状態にたどり着くと値を返して止まる。

Code 4.3 は Code 4.1 で書いた While Loop を Hoare Logic での Comm で記述したものである。ここでの $ は () の対応を合わせる Agda の糖衣構文で、行頭から行末までを () で囲っていることと同義である。

ソースコード 4.3: Hoare Logic のプログラム

```
1  program : Comm
2  program =
3    Seq ( PComm (λ env → record env {varn = 10}))
4      $ Seq ( PComm (λ env → record env {vari = 0}))
5      $ While (λ env → lt zero (varn env ) )
6        (Seq (PComm (λ env → record env {vari = ((vari env) + 1)} ))
7          $ PComm (λ env → record env {varn = ((varn env) - 1)} ))
```

この Comm は Command をならべているだけである。この Comm を Agda 上で実行するため、 Code 4.4 のような interpreter を記述した。

ソースコード 4.4: Agda での Hoare Logic interpreter

```
{-# TERMINATING #-}
interpret : Env → Comm → Env
interpret env Skip = env
interpret env Abort = env
interpret env (PComm x) = x env
interpret env (Seq comm comm1) = interpret (interpret env comm) comm1
interpret env (If x then else) with x env
... | true = interpret env then
... | false = interpret env else
interpret env (While x comm) with x env
... | true = interpret (interpret env comm) (While x comm)
... | false = env
```

Code 4.4 は 初期状態の Env と 実行する Command の並びを受けとって、実行後の Env を返すものとなっている。

ソースコード 4.5: Agda での Hoare Logic の実行

```
test : Env
test =  interpret ( record { vari = 0  ; varn = 0 } ) program
```

Code 4.5 のように interpret に $vari = 0, varn = 0$ の record を渡し、 実行する Comm を渡して 評価してやると $recordvarn = 0; vari = 10$ のような Env が返ってくる。

## 4.2 Hoare Logic での部分正当性

ここでは先程例とした **??** の部分正当性の検証を行う。

Code **??** は Agda 上での Hoare Logic での Command の検証である。HTProof では Condition と Command もう一つ Condition を受け取って、 Set を返す Agda のデータとして表現されている。

PrimRule は Code 4.6 の Axiom という関数を使い、事前条件が成り立っている時、実行後に事後条件が成り立つならば、 PComm で変数に値を代入できることを保証している。

SkipRule は Condition を受け取ってそのままの Condition を返すことを保証する。

AbortRule は PreContition を受け取って、Abort を実行して終わるルールである。

WeakeningRule は 4.6 の Tautology という関数を使って通常の逐次処理から、WhileRule のみに適応されるループ不変変数に移行する際のルールである。

SeqRule は 3 つの Condition と 2 つの Command を受け取り、これらのプログラムの逐次的な実行を保証する。

IfRule は分岐に用いられ、3 つの Condition と 2 つの Command を受け取り、判定の Condition が成り立っているかいないかで実行する Command を変えるルールである。この時、どちらかの Command が実行されることを保証している。

WhileRule はループに用いられ、1 つの Command と 2 つの Condition を受け取り、事前条件が成り立っている間、 Command を繰り返すことを保証している。

ソースコード 4.6: Axiom と Tautology

```
_⇒_ : Bool → Bool → Bool
false  ⇒  _ = true
true  ⇒  true = true
true  ⇒  false = false

Axiom : Cond → PrimComm → Cond → Set
Axiom pre comm post = ∀ (env : Env) →  (pre env) ⇒ ( post (comm env))
   ≡ true

Tautology : Cond → Cond → Set
Tautology pre post = ∀ (env : Env) →  (pre env)  ⇒ (post env) ≡ true
```

Code **??**を使って Code 5.1 の whileProgram の仕様を構成する。

ソースコード 4.7: Agda での Hoare Locig の構成

```
data HTProof : Cond → Comm → Cond → Set where
  PrimRule : {bPre : Cond} → {pcm : PrimComm} → {bPost : Cond} →
             (pr : Axiom bPre pcm bPost) →
             HTProof bPre (PComm pcm) bPost
  SkipRule : (b : Cond) → HTProof b Skip b
  AbortRule : (bPre : Cond) → (bPost : Cond) →
              HTProof bPre Abort bPost
  WeakeningRule : {bPre : Cond} → {bPre' : Cond} → {cm : Comm} →
                  {bPost' : Cond} → {bPost : Cond} →
                  Tautology bPre bPre' →
                  HTProof bPre' cm bPost' →
                  Tautology bPost' bPost →
                  HTProof bPre cm bPost
  SeqRule : {bPre : Cond} → {cm1 : Comm} → {bMid : Cond} →
            {cm2 : Comm} → {bPost : Cond} →
            HTProof bPre cm1 bMid →
            HTProof bMid cm2 bPost →
            HTProof bPre (Seq cm1 cm2) bPost
  IfRule : {cmThen : Comm} → {cmElse : Comm} →
           {bPre : Cond} → {bPost : Cond} →
           {b : Cond} →
           HTProof (bPre /\ b) cmThen bPost →
           HTProof (bPre /\ neg b) cmElse bPost →
           HTProof bPre (If b cmThen cmElse) bPost
  WhileRule : {cm : Comm} → {bInv : Cond} → {b : Cond} →
              HTProof (bInv /\ b) cm bInv →
              HTProof bInv (While b cm) (bInv /\ neg b)
```

全体の仕様は Code 4.8 の proof1 の様になる。proof1 では型で initCond、 Code 4.3 の program、 termCond を記述しており、initCond から program を実行し termCond に行き着く Hoare Logic の証明になる。

それぞれの Condition は Rule の後に記述されている  に囲まれた部分で、initCond のみ無条件で true を返す Condition になっている。

それぞれの Rule の中にそこで証明する必要のある補題が lemma で埋められている。lemma1 から lemma5 の証明は幅を取ってしまうため、 全体は付録に載せる。

これらの lemma は HTProof の Rule に沿って必要なものを記述されており、lemma1 では PreCondition と PostCondition が存在するときの代入の保証、lemma2 では While Loop に入る前の Condition からループ不変条件への変換の証明、lemma3 では While Loop 内での PComm の代入の証明、lemma4 では While Loop を抜けたときの Condition の整合性、lemma5 では While Loop を抜けた後のループ不変条件から Condition への変換と termCond への移行の整合性を保証している。

ソースコード 4.8: Agda 上での WhileLoop の検証

```
1 proof1 : HTProof initCond program termCond
2 proof1 =
3   SeqRule {λ e → true} ( PrimRule empty-case )
4     $ SeqRule {λ e →  Equal (varn e) 10} ( PrimRule lemma1   )
5     $ WeakeningRule {λ e → (Equal (varn e) 10) ∧ (Equal (vari e) 0)}
    lemma2 (
6       WhileRule {_} {λ e → Equal ((varn e) + (vari e)) 10}
7         $ SeqRule (PrimRule {λ e →  whileInv e  ∧ lt zero (varn e) }
    lemma3 )
8         $ PrimRule {whileInv'} {_} {whileInv}  lemma4 ) lemma5
```

proof1 は Code 4.3 の program と似た形をとっている。Hoare  Logic では Comannd に対応する証明規則があるため、仕様はプログラムに対応している。

## 4.3 Hoare Logic での健全性

4.8 では Agda での Hoare Logic を用いた仕様の構成を行った。この仕様が実際に正しく動作するかどうか (健全性) を証明する必要がある。

4.9 は Hoare Logic 上での部分正当性を確かめるための関数である。SemComm では Comm を受け取って成り立つ関係を返す。Satisfies では Pre Condition と Command、Post Condition を受け取って、Pre Condition から Post Condition を正しく導けるという仕様を返す。

ソースコード 4.9: State Sequence の部分正当性

```
1 SemComm : Comm → Rel State (Level.zero)
2 SemComm Skip = RelOpState.deltaGlob
```

```
SemComm Abort = RelOpState.emptyRel
SemComm (PComm pc) = PrimSemComm pc
SemComm (Seq c1 c2) = RelOpState.comp (SemComm c1) (SemComm c2)
SemComm (If b c1 c2)
  = RelOpState.union
    (RelOpState.comp (RelOpState.delta (SemCond b))
                     (SemComm c1))
    (RelOpState.comp (RelOpState.delta (NotP (SemCond b)))
                     (SemComm c2))
SemComm (While b c)
  = RelOpState.unionInf
    (λ (n : $mathbb{N}$) →
      RelOpState.comp (RelOpState.repeat
        n
        (RelOpState.comp
          (RelOpState.delta (SemCond b))
          (SemComm c)))
      (RelOpState.delta (NotP (SemCond b))))

Satisfies : Cond → Comm → Cond → Set
Satisfies bPre cm bPost
  = (s1 : State) → (s2 : State) →
    SemCond bPre s1 → SemComm cm s1 s2 → SemCond bPost s2
```

これらの仕様を検証することでそれぞれの Command に対する部分正当性を示す。

4.10 の Soundness では HTProof を受け取り、Satisfies に合った証明を返す。Soundness では HTProof に記述されている Rule でパターンマッチを行い、対応する証明を適応している。

ソースコード 4.10: Agda での Hoare Logic の健全性

```
Soundness : {bPre : Cond} → {cm : Comm} → {bPost : Cond} →
            HTProof bPre cm bPost → Satisfies bPre cm bPost
Soundness (PrimRule {bPre} {cm} {bPost} pr) s1 s2 q1 q2
  = axiomValid bPre cm bPost pr s1 s2 q1 q2
Soundness {.bPost} {.Skip} {bPost} (SkipRule .bPost) s1 s2 q1 q2
  = substId1 State {Level.zero} {State} {s1} {s2} (proj₂ q2) (SemCond
    bPost) q1
Soundness {bPre} {.Abort} {bPost} (AbortRule .bPre .bPost) s1 s2 q1 ()
Soundness (WeakeningRule {bPre} {bPre'} {cm} {bPost'} {bPost} tautPre pr
    tautPost)
          s1 s2 q1 q2
  = let hyp : Satisfies bPre' cm bPost'
        hyp = Soundness pr
        r1 : SemCond bPre' s1
        r1 = tautValid bPre bPre' tautPre s1 q1
        r2 : SemCond bPost' s2
        r2 = hyp s1 s2 r1 q2
    in tautValid bPost' bPost tautPost s2 r2
Soundness (SeqRule {bPre} {cm1} {bMid} {cm2} {bPost} pr1 pr2)
           s1 s2 q1 q2
  = let hyp1 : Satisfies bPre cm1 bMid
        hyp1 = Soundness pr1
```

```
21          hyp2 : Satisfies bMid cm2 bPost
22          hyp2 = Soundness pr2
23          sMid : State
24          sMid = proj_1 q2
25          r1 : SemComm cm1 s1 sMid × SemComm cm2 sMid s2
26          r1 = proj_2 q2
27          r2 : SemComm cm1 s1 sMid
28          r2 = proj_1 r1
29          r3 : SemComm cm2 sMid s2
30          r3 = proj_2 r1
31          r4 : SemCond bMid sMid
32          r4 = hyp1 s1 sMid q1 r2
33      in hyp2 sMid s2 r4 r3
34  Soundness (IfRule {cmThen} {cmElse} {bPre} {bPost} {b} pThen pElse)
35            s1 s2 q1 q2
36    = let hypThen : Satisfies (bPre /\ b) cmThen bPost
37          hypThen = Soundness pThen
38          hypElse : Satisfies (bPre /\ neg b) cmElse bPost
39          hypElse = Soundness pElse
40          rThen : RelOpState.comp
41                    (RelOpState.delta (SemCond b))
42                    (SemComm cmThen) s1 s2 →
43                  SemCond bPost s2
44          rThen = λ h →
45                    let t1 : SemCond b s1 × SemComm cmThen s1 s2
46                        t1 = (proj_2 (RelOpState.deltaRestPre
47                                        (SemCond b)
48                                        (SemComm cmThen) s1 s2)) h
49                        t2 : SemCond (bPre /\ b) s1
50                        t2 = (proj_2 (respAnd bPre b s1))
51                             (q1 , proj_1 t1)
52                    in hypThen s1 s2 t2 (proj_2 t1)
53          rElse : RelOpState.comp
54                    (RelOpState.delta (NotP (SemCond b)))
55                    (SemComm cmElse) s1 s2 →
56                  SemCond bPost s2
57          rElse = λ h →
58                    let t10 : (NotP (SemCond b) s1) ×
59                              (SemComm cmElse s1 s2)
60                        t10 = proj_2 (RelOpState.deltaRestPre
61                                        (NotP (SemCond b)) (SemComm cmElse)
      s1 s2)
62                                h
63                        t6 : SemCond (neg b) s1
64                        t6 = proj_2 (respNeg b s1) (proj_1 t10)
65                        t7 : SemComm cmElse s1 s2
66                        t7 = proj_2 t10
67                        t8 : SemCond (bPre /\ neg b) s1
68                        t8 = proj_2 (respAnd bPre (neg b) s1)
69                             (q1 , t6)
70                    in hypElse s1 s2 t8 t7
71      in when rThen rElse q2
72  Soundness (WhileRule {cm'} {bInv} {b} pr) s1 s2 q1 q2
73    = proj_2 (respAnd bInv (neg b) s2) t20
```

```
74      where
75        hyp : Satisfies (bInv /\ b) cm' bInv
76        hyp = Soundness pr
77        n : $mathbb{N}$
78        n = proj_1 q2
79        Rel1 : $mathbb{N}$ → Rel State (Level.zero)
80        Rel1 = λ m →
81                 RelOpState.repeat
82                   m
83                   (RelOpState.comp (RelOpState.delta (SemCond b))
84                                    (SemComm cm'))
85        t1 : RelOpState.comp
86               (Rel1 n)
87               (RelOpState.delta (NotP (SemCond b))) s1 s2
88        t1 = proj_2 q2
89        t15 : (Rel1 n s1 s2) × (NotP (SemCond b) s2)
90        t15 = proj_2 (RelOpState.deltaRestPost
91                       (NotP (SemCond b)) (Rel1 n) s1 s2)
92                 t1
93        t16 : Rel1 n s1 s2
94        t16 = proj_1 t15
95        t17 : NotP (SemCond b) s2
96        t17 = proj_2 t15
97        lem1 : (m : $mathbb{N}$) → (ss2 : State) → Rel1 m s1 ss2 →
98               SemCond bInv ss2
99        lem1 $mathbb{N}$.zero ss2 h
100         = substId1 State (proj_2 h) (SemCond bInv) q1
101       lem1 ($mathbb{N}$.suc n) ss2 h
102         = let hyp2 : (z : State) → Rel1 n s1 z →
103                      SemCond bInv z
104               hyp2 = lem1 n
105               s20 : State
106               s20 = proj_1 h
107               t21 : Rel1 n s1 s20
108               t21 = proj_1 (proj_2 h)
109               t22 : (SemCond b s20) × (SemComm cm' s20 ss2)
110               t22 = proj_2 (RelOpState.deltaRestPre
111                               (SemCond b) (SemComm cm') s20 ss2)
112                     (proj_2 (proj_2 h))
113               t23 : SemCond (bInv /\ b) s20
114               t23 = proj_2 (respAnd bInv b s20)
115                     (hyp2 s20 t21 , proj_1 t22)
116           in hyp s20 ss2 t23 (proj_2 t22)
117       t20 : SemCond bInv s2 × SemCond (neg b) s2
118       t20 = lem1 n s2 t16 , proj_2 (respNeg b s2) t17
```

4.11 は HTProof で記述された仕様を、実際に構成可能な仕様を満たしているかを確認する Satisfies を返す。照明部分では HTProof で構成された使用を受け取り、 Soundness が対応した証明を返すようになっている。

ソースコード 4.11: ?

```
1 PrimSoundness : {bPre : Cond} -> {cm : Comm} -> {bPost : Cond} ->
```

```
2             HTProof bPre cm bPost -> Satisfies bPre cm bPost
3 PrimSoundness {bPre} {cm} {bPost} ht = Soundness ht
```

4.12 は 4.3 の program の Hoare Logic での証明である。この証明では初期状態 *initCond* と実行するコマンド群 *program* を受け取り終了状態として *termCond* が true であることを示す。

ソースコード 4.12: while program の健全性

```
1 proofOfProgram : (c10 : $mathbb{N}$) → (input output : Env )
2   → initCond input ≡ true
3   → (SemComm (program c10) input output)
4   → termCond {c10} output ≡ true
5 proofOfProgram c10 input output ic sem  = PrimSoundness (proof1 c10)
    input output ic sem
```

この証明は実際に構築した仕様である *proof*1 を *PrimSoundness* に入力することで満たすことができる。ここまで記述することで Agda 上の Hoare Logic を用いた while program を検証することができた。

# 第5章　Continuation based C と Agda

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。

先行研究?? では CbC と Agda を対応させるための型付けが行われているが、ここでは、その型付けは使わず、前段階である Agda での記述のみで説明を行う。

本章では当研究室で推奨している単位での検証を行うために、Agda で DataGear、CodeGear を表現し、これらの単位を用いた検証を行う事ができることを示す。

## 5.1　DataGear、 CodeGear と Agdaの対応

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ので、Context を通すことなくそのまま扱う。

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。コード 5.1 は Agda で記述した CodeGear の例である。

ソースコード 5.1: whileTest の型

```
1 whileTest : {l : Level} {t : Set l} → (c10 : ℕ) → (Code : Env → t) →
      t
2 whileTest c10 next = next (record {varn = c10 ; vari = 0} )
```

型では c10 と名付けた自然数を受け取った後、Env を受け取り不定の型 t を返す関数を受け取り、最終的に t を返す CodeGear を定義しており、

実装では、自然数 c10 と継続先の関数 next を受け取り、next に値を代入した Env を引数として渡している。

## 5.2　Meta Gears の表現

通常の Meta Gears はノーマルレベルの CodeGear、 DataGear では扱えないメタレベルの計算を扱う CodeGear である。検証での Meta DataGear は、 DataGear が持つ同値関係や、大小関係などの関係を表す DataGear がそれに当たると考えている。Agda 上では Meta DataGear を持つことで性質を持ったデータ構造を作ることができる。

Meta CodeGear は 通常の CodeGear では扱えないメタレベルの計算を扱う CodeGear である。Agda での Meta CodeGear は通常の CodeGear を引数に取りそれらの関係などの上位概念を返す CodeGear である。これは (図を入れる) のような Code Gear となる。

## 5.3　CbC 上での HoareLogic の実現

CbC 上の Hoare Logic は引数として事前条件、次の CodeGear に渡す値に事後条件を含めることで記述する。その際に事前条件が CodeGear で変更され、事後条件を導く形になる。例として while プログラムの CbC 記述についてみる。

ここでは

Hoare Logic の記述を行い、部分的な整合性を示すことができている。全体の検証を行うためには接続されているすべての CodeGear が実行されたときの健全性 (Soundness) が担保される必要がある。そのため、検証用の Meta CodeGear を記述する。例として while プログラムの健全性を担保するプログラムをみる。このコードでは CodeGear をつなげて終了状態まで実行したとき最後の事後条件が成り立っているため、これらの実行が正しく終了することを示すことができる。

# 第6章　CbC と Hoare Logic

5 では CbC の CodeGear、 DataGear という記述の Agda への対応を示し、CbC で書かれたプログラムが検証できることを確認した。また、4 では Agda 上での Hoare Logic を用いて検証を行った。

6 では CbC での CodeGear、 DataGear という記述と Hoare Logic を対応させ、Hoare Logic をベースとした CbC の検証手法を定義する。さらに Hoare Logic で例とした while program に対して同様に検証を行う。

## 6.1　CbC 上での Hoare Logic の文法

Hoare Logic では事前条件、計算、事後条件があり、計算によって事前条件から事後条件を導くことで部分的な正当性を導くことができた。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ことで HoareLogic と同様の構造にすることができる。

6.2 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。

ソースコード 6.1: CbC 上での Hoare Logic

```
1  -- 通常の CodeGear
2  whileLoop' : {l : Level} {t : Set l} → (n : ℕ) → (env : Envc) → (n ≡
       varn env) → (next : Envc → t) → (exit : Envc → t) → t
3  whileLoop' zero env refl _ exit = exit env
4  whileLoop' (suc n) env refl next _ = next (record env {varn = pred (varn
       env) ; vari = suc (vari env) })
5
6  -- Hoare Logic ベースの CodeGear
7  whileLoopPwP' : {l : Level} {t : Set l} → (n : ℕ) → (env : Envc ) → (n
       ≡ varn env) → (pre : varn env + vari env ≡ c10 env)
8    → (next : (env : Envc ) → (pred n ≡ varn env) → (post : varn env +
       vari env ≡ c10 env) → t)
9    → (exit : (env : Envc ) → (fin : vari env ≡ c10 env)  → t) → t
10 whileLoopPwP' zero env refl refl next exit = exit env refl
```

```
11 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。

また、Hoare Logic では HTProof というコマンドと対応した公理が存在していたが、CbC では各 CodeGear に対応した事前、事後条件付きの Meta CodeGear を記述することがそれに当たる。

## 6.2　CbC 上での Hoare Logic の仕様記述

## 6.3　CbC 上での Hoare Logic を用いた仕様記述と検証

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。

ソースコード 6.2: CbC 上での Hoare Logic

```
1  -- 通常の CodeGear
2  whileLoop' : {l : Level} {t : Set l} → (n : ℕ) → (env : Envc) → (n ≡
       varn env) → (next : Envc → t) → (exit : Envc → t) → t
3  whileLoop' zero env refl _ exit = exit env
4  whileLoop' (suc n) env refl next _ = next (record env {varn = pred (varn
       env) ; vari = suc (vari env) })
5
6  -- Hoare Logic ベースの CodeGear
7  whileLoopPwP' : {l : Level} {t : Set l} → (n : ℕ) → (env : Envc ) → (n
       ≡ varn env) → (pre : varn env + vari env ≡ c10 env)
8    → (next : (env : Envc ) → (pred n ≡ varn env) → (post : varn env +
       vari env ≡ c10 env) → t)
9    → (exit : (env : Envc ) → (fin : vari env ≡ c10 env)  → t) → t
10 whileLoopPwP' zero env refl refl next exit = exit env refl
11 whileLoopPwP' (suc n) env refl refl next exit = next (record env {varn =
       pred (varn env) ; vari = suc (vari env) }) refl (+-suc n (vari env))
```

*whileTestPCallwP′* は while program と同様の動作をする CodeGear を組みあわせた仕様である。*whileTestPwP* では任意の自然数 $c$ を受け取り、vari、 varn、 c10 の 3 つの変数を保持する DataGear である env にそれぞれ代入を行い、env と env に意図した代入が行われていることを示す Meta DataGear $s$ を次の関数に渡している。この Meta DataGear $s$ は *whileTestPwP* の事後条件に当たる。

*loopPwP′* は *whileTestPwP* に変更された env と Meta DataGear $s$ を受け取り、関数内でループを行う。*loopPwP′* では無限ループを避けるためループに自然数の減少を絡

め有限回で停止するよう工夫をしている。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。

これらの関数を実行したとき、最後のラムダ式に入っている最終状態 $varienv \equiv c10env$ が必ず成り立つという仕様になっている。

$whileTestPCallwP'$ を検証するには、?? のように $\mathbb{N}$ を受け取って $whileTestPCallwP'\mathbb{N}$ が成り立つ型を記述し、実際に導出部分で定義してやれば良い。$whileTestPwP$ は代入のため単純に Agda が計算できるが、 $loopPwP'$ などのループは実際の値が入るまで計算をすすめることができない。そのため、 loop を簡約する補助定理 $loopHelper$ を別に用意した。

$loopHelper$ では $loopPwP'$ を実際に実行したとき、 $varienv \equiv c10env$ が成り立つことを証明している。

$loopHelper$ を使い $loopPwP$ を簡約し、$whileSoundness$ の導出をすることができた。

# 第7章　結論

まだ終わってないので最後に

## 7.1　今後の課題

いつかくる後輩の修論や卒論に備えて

# 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。

2020 年 3 月
外間 政尊

# 参考文献


[1] 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*, Vol. 53, No. 6, pp. 107–115, June 2010.

[2] 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, pp. 252–269, New York, NY, USA, 2017. ACM.

[3] Ats-pl-sys. `http://www.ats-lang.org/`. Accessed: 2018/12/17(Mon).

[4] Rust programming language. `https://www.rust-lang.org/`. Accessed: 2018/12/17(Mon).

[5] 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.

[6] Welcome! — the coq proof assistant. `https://coq.inria.fr/`. Accessed: 2018/12/17(Mon).

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

[8] 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.

[9] 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.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, Vol. 12, No. 10, p. 576–580, October 1969.

[11] Example - hoare logic. `http://ocvs.cfv.jp/Agda/readmehoare.html`. Accessed: 2018/12/17(Mon).

[12] Hoare logic in agda2. `https://github.com/IKEGAMIDaisuke/HoareLogic`. Accessed: 2018/12/17(Mon).

[13] 比嘉健太, 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）, Vol. 10, No. 2, pp. 5–5, feb 2017.

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

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

[16] 政尊外間, 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.

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

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

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

[20] 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: 2018/12/17(Mon).

[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, 2017.

[22] 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, 2018.