修士 (工学) 学位論文

Master's Thesis of Engineering

GearsOS での HoareLogic を用いた実装と検証

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)　　和田　知久

印

(副　査)　　名嘉村　盛和

印

(副　査)　　長田　智和

印

(副　査)　　河野　真治

要旨

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。

Abstract

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

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[**?**] 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[?] (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プログラムを記述することになる。CbC でのプログラミングは DataGear を CodeGear で変更し、その変更を次の CodeGear に渡して処理を実行する。現在 CbC の処理系には llvm/clang による実装 [?] と gcc[?] による実装が存在している。

本章は CbC について説明する。

2.1　Code Gear と Data Gear

本研究室では検証しやすいプログラムの単位として CodeGear と DataGear 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。

CodeGear　とは処理を行う単位である。

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。

図 2.1 のように Input DataGear を受け取り、CodeGear で処理を行い、Output DataGear に変更を加え、プログラム全体を記述する。

図 2.1: CodeGear と DataGear

2.2　メタ計算

メタ計算 (自己反映計算)[?] とはプログラムを記述する際に通常の処理と分離し、他に記述しなければならない処理である。例えばプログラム実行時のメモリ管理やスレッド管

理、資源管理等の計算がこれに当たる。

メタ計算は関数型言語では Monad[7] を用いて表現される [?]。Monad は Haskell では実行時の環境を記述する構文として使われる。

従来の OS では、メタ計算はシステムコールやライブラリーコールの単位で行われる。実行時にメタ計算の変更を行う場合には OS 内部のパラメータの変更を使用し、実行されるユーザープログラム自体への変更は限定的である。しかし、メタ計算は性能測定あるいはプログラム検証、さらに並列分散計算のチューニングなど細かい処理が必要で実際のシステムコール単位では不十分である。例えば、モデル検査ではアセンブラあるいは バイトコード、インタプリタレベルでのメタ計算が必要になる。しかし、バイトコードレベルでは 粒度が細かすぎて扱いが困難になっている。具体的にはメタ計算の実行時間が大きくなってしまう。

2.3 Context

CbC では、接続可能な CodeGear、DataGear のリスト、Temporal DataGear のためのメモリ空間などを Context として保持している。CbC で必要な CodeGear 、 DataGear を参照する際は Context を通してアクセスする。

2.4 Meta Gears

Meta Gear は CbC 上でのメタ計算である。CodeGear を実行する際、必要な DataGear を Context から取得する必要がある。しかし、 ユーザーが Context から直接データを扱える状態は信頼性を欠く。そのため、 CbC では 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 Gear を定義している。

第3章　Agda

Adga [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っており、型を第一級オブジェクトとして扱うことができる。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について説明を行う。また、定理証明支援器としての Agda について解説する。

3.1　Agda の文法

Agda 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` や `{-- comment --}` のように記述される。

Agda のプログラムは全てモジュール内部に記述されるため、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である。

通常の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リスト 3.1 に示す。

ソースコード 3.1: Agda におけるモジュールのインポート

```
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 における型指定は : を用いて行う。 変数 x が型 A を持つ、ということを表すには x : A と記述する。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た後、値にコンストラクタとその型を列挙する。

例えば Bool 型を定義するとリスト 3.2 のようになる。Bool はコンストラクタ `true` と `false` を持つデータ型である。Bool 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

ソースコード 3.2: Agda におけるデータ型 Bool の定義

```
1 data Bool : Set where
2   true  : Bool
3   false : Bool
```

Agda には C 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` を定義するとリスト 3.3 のようになる。レコードを構築する際は `record` キーワードの後の `{}` の内部に `fieldName = value` の形で値を列挙していく。複数の値を列挙する際は ; で区切る。

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 }
```

3.3 Agda の関数

Agda での関数の定義は、関数名と型を記述した後に関数の本体を = の後に記述する。関数の型には → 、 または-> を用いる。

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。Bool 変数 `x` を取って true を返す関数 `f` はリスト 3.4 のようになる。

ソースコード 3.4: Agda における関数定義

```
1 f : Bool -> Bool
2 f x = true
```

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例えば Bool 型の値を反転する not 関数を書くとリスト 3.5 のようになる。

ソースコード 3.5: Agda における関数 not の定義

```
1 not : Bool -> Bool
2 not  true = false
3 not false = true
```

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まなくてはならない。例えば、Bool 型を受け取る関数で true の時の挙動のみを書くことはできない。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と、変数が持ちうる値は指定した以外のコンストラクタとなる。例えばリスト 3.6 の not は x には true しか入ることは無い。なお、マッチした値以外の挙動をまとめて書く際には _ を用いることもできる。

ソースコード 3.6: Agda におけるパターンマッチ

```
1 not : Bool -> Bool
2 not false = true
3 not x     = false
```

Agda にはラムダ計算が存在している。ラムダ計算とは関数内で生成できる無名の関数であり、\arg1 arg2 -> function body のように書くことができる。例えば Bool 型の引数 b を取って not を適用する not-apply をラムダ計算で書くとリスト 3.7 のようになる。関数 not-apply をラムダ計算を使わずに定義すると not-apply-2 になるが、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 not-apply : Bool -> Bool
2 not-apply = (\b -> not b)   -- use lambda
3
4 not-apply : Bool -> Bool
5 not-appy b = not b          -- not 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
```

```
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)
```

3.4 定理証明支援器としての Agda

Agda での証明では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が完成する。証明の例として Code 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。$y = sucy$ の時は x == y の時 fx == 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 $+zeroy$ を用いて証明している。

ソースコード 3.9: 等式変形の例

```
+zero : { y : ℕ } → y + zero  ≡ y
+zero {zero} = refl
+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存在している。Code 3.10、3.12 は等式変形の例である。始めに等式変形を始めたいところで $letopen \equiv -Reasoninginbegin$ と記述する。Agda 上では分からないところを ? と置いておくことができるので、残りを ? としておく。−− は Agda 上ではコメントである。

ソースコード 3.10: 等式変形の例 1/2

```
stmt2Cond : {c10 :ℕ} → Cond
stmt2Cond {c10} env = (Equal (varn env) c10) ∧ (Equal (vari env) 0)

lemma1 : {c10 :ℕ} → Axiom (stmt1Cond {c10}) (λ env →
     record { varn = varn env ; vari = 0 }) (stmt2Cond {c\
10})
lemma1 {c10} env = impl⇒ ( λ cond → let open ≡-Reasoning  in
begin
?       -- ?0
≡⟨ ? ⟩  -- ?1
?       -- ?2
■ )

-- ?0 : Bool
-- ?1 : stmt2Cond (record { varn = varn env ; vari = 0 }) ≡ true
-- ?2 : Bool
```

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 $x == y$ のような変形規則を入れることで等式を変形して証明することができる。

ここでは 3.11 の Bool 値 x を受け取って $x \wedge true$ の時必ず x であるという証明 ∧true と 値と Env を受け取って Bool 値を返す stmt1Cond を使って等式変形を行う。

ソースコード 3.11: 使っている等式変形規則

```
1  ∧true : { x : Bool } →  x  ∧  true  ≡ x
2  ∧true {x} with x
3  ∧true {x} | false = refl
4  ∧true {x} | true = refl
5
6  stmt1Cond : {c10 :ℕ} → Cond
7  stmt1Cond {c10} env = Equal (varn env) c10
```

最終的な証明は 3.12 のようになる。

ソースコード 3.12: 等式変形の例 2/2

```
1  lemma1 : {c10 :ℕ} → Axiom (stmt1Cond {c10}) (λ env →
       record { varn = varn env ; vari = 0 }) (stmt2Cond {c\
2  10})
3  lemma1 {c10} env = impl⇒ ( λ cond → let open ≡-Reasoning  in
4  begin
5  (Equal (varn env) c10 ) ∧ true
6  ≡⟨ ∧true ⟩
7  Equal (varn env) c10
8  ≡⟨ cond ⟩
9  true
10 ■ )
```

3.5 定理証明とプログラミング検証

たぶん Curry-Howard isomorphism の話

第4章　Floyd-Hoare Logic

HoareLogic[?] とは C.A.R Hoare、 R.W Floyd らによるプログラムの検証の手法である。HoareLogic では事前条件 (Pre-Condition) が成り立つとき、何らかの計算 (Command) を実行した後に事後条件 (Post-Condition) 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。この PCQ のことを HoareTriple と呼ぶ。HoareTriple 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、Q のあとに別の C をつなげてプログラムを構築し、すべての実行を検証することができる。

4.1　Agda での Hoare Logic システムの構築

現在 Agda 上での HoareLogic は初期の Agda で実装されたもの [8] とそれを現在の Agda に対応させたもの [9] が存在している。

例として Agda に対応させたもの [9] の Command と証明のためのルールを使って HoareLogic を実装した。Code 4.2 は Agda 上での HoareLogic の構築子である。ここでの Comm は Agda2 に対応した Command の定義を使用している。

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 HoareLogic 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 HoareLogic 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 HoareLogic 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 HoareLogic 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 HoareLogic 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: HoareLogic 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 HoareLogic 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 HoareLogic 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 Agda 上での Hoare Logic システムの検証

ここでは先程例とした 4.1 の検証を例とする。

Code 4.7 は Agda 上での HoareLogic での証明の構成である。HTProof では Condition と Command もう一つ Condition を受け取って、 Set を返す Agda のデータである。ここでの HTProof [9] も Agda2 に移植されたものを使っている。

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
```

ソースコード 4.7: Agda での HoareLogic 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)
```

Code 4.7 を使って Code 4.1 の whileProgram の証明を構築する。

全体の証明は 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 HoareLogic 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、 詳細は当研究室レポジトリ [10] のプログラムを参照していただきたい。

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。HoareLogic では Comannd に対応する証明規則があるため、証明はプログラムに対応している。

4.3 Agda 上での Hoare Logic システムの健全性

4.8 では Agda での HoareLogic を用いた証明の構築を行った。システムの健全性は本来 Agda 側で保証する必要があるが、今回の例では Agda 上に Hoare Logic を行えるシステムを構築したので健全性を担保する必要がある。hogefugapiyo

第5章　Continuation based C と Agda

当研究室で推奨されている CodeGear、 DataGear という単位で検証を行うため、Agda 上で CodeGear、 DataGear という単位を対応させる。

5.1　DataGear の対応

CbC では DataGear はすべての CodeGear から参照できる record として context 記述される。Agda 上でも record 型のデータを使い記述することができるが...

5.2　CodeGear

CodeGear は DataGear を受け取って DataGear を返すという定義であるため、Continuation Passing Style で書かれた Agda の関数と対応する。

CodeGear の実行は CodeGear 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。

5.3　Meta CodeGear の表現

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。

5.4　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 の記述としてはこれで良く、部分整合性は示せているが、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　BinaryTree

CbC-Agda 上での Binary Tree

6.1　BinaryTree

BinaryTree 概要

6.2　BinaryTree の実現

BinaryTree の記述等

6.3　BinaryTree の検証時の問題点と改善

つまってたとこ (条件付きとかそのあたり)

6.4　BinaryTree 検証の HoareLogic を用いた解決

できたらいいなの話 (1/2 時点)

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。

2018 年 3 月
伊波立樹

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

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

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

[10] 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).

[11] 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.

[12] 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.

[13] 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.

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

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

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