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.

# GearsOS の Hoare Logic をベースにした検証手法

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　あらまし
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. まえがき

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。HoareLogic はシンプルなアプローチだ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。

## 2. 現　　状

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 [9], [10] 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 [2] や Rust [5] 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 [1]、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 [12] 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。

## 3. Agda

Adga [1] とは証明支援系の関数型言語である。Agda におけ

る型指定は：を用いて行う。例えば、 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。 Code 1 はこの data 型の例である。Comm では Skip、Abort、PComm などのコンストラクタを持ち、: の後に型が書かれている。

Code 1: data の例

```
data Comm : Set where
  Skip  : Comm
  Abort : Comm
  PComm : PrimComm -> Comm
  Seq   : Comm -> Comm -> Comm
  If    : Cond -> Comm -> Comm -> Comm
  While : Cond -> Comm -> Comm
```

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、record キーワード後の内部に field キーワードを記述し、その下に Name = value の形で値を列挙していく。複数の値を列挙する際は ; で区切る必要がある。Code 2 はレコード型の例であり、Env では varn と vari の二つの field を持ち、それぞれの型が Agda 上で自然数を表す Nat になっている。

Code 2: record の例

```
record Env : Set where
  field
    varn : ℕ
    vari : ℕ
```

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 のように書ける。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。ここでは Code 3 を例にとる。これは引き算の演算子で、Agda の Nat を二つ引数に受けて Nat を返す関数である。

Code 3: 関数の例

```
_−_ :ℕ → ℕ → ℕ
x − zero  = x
zero − _  = zero
(suc x) − (suc y)  = x − y
```

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 4 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。

Code 4: 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 5、 7 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。

Code 5: 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 6 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。

Code 6: 使っている等式変形規則

```
∧true : { x : Bool } →  x  ∧  true  ≡ x
∧true {x} with x
∧true {x} | false = refl
∧true {x} | true = refl

stmt1Cond : {c10 :ℕ} → Cond
stmt1Cond {c10} env = Equal (varn env) c10
```

最終的な証明は 7 のようになる。

Code 7: 等式変形の例 2/2

```
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
4  begin
5  (Equal (varn env) c10 ) ∧ true
6  ≡⟨ ∧true ⟩
7  Equal (varn env) c10
8  ≡⟨ cond ⟩
9  true
10 ■ )
```

## 4. GearsOS

Gears OS は信頼性をノーマルレベルの計算に対して保証し、拡張性をメタレベルの計算で実現することを目標に開発している OS である。

Gears OS は処理の単位を Code Gear、データの単位を Data Gear と呼ばれる単位でプログラムを構成する。信頼性や拡張性はメタ計算として、通常の計算とは区別して記述する。Gears OS は Code Gear、Data Gear を用いたプログラミング言語である CbC(Continuation based C) で実装される。そのため、Gears OS の実装は Gears を用いたプログラミングスタイルの指標となる。

## 5. CodeGear と DataGear

Gears OS ではプログラムとデータの単位として CodeGear、DataGear を用いる。 Gear は並列実行の単位、データ分割、Gear 間の接続等になる。CodeGear はプログラムの処理そのもので、図 Code 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。

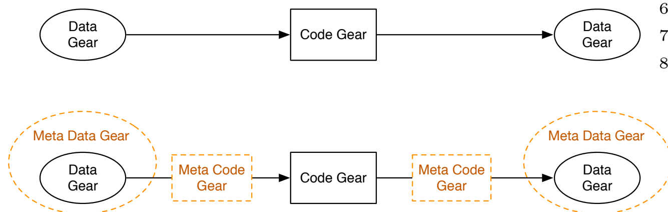


Figura 1: CodeGear と DataGear の関係

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。

## 6. Gears と Agda

ここでは Gears を用いたプログラムを検証するため、Agda 上での CodeGear、 DataGear との対応をみていく。

CodeGear は Agda では継続渡しで書かれた関数と等価である。継続は不定の型 (t) を返す関数で表される。CodeGear 自体も同じ型 t を返す関数となる。このとき、継続に渡される引数や、関数から出力される値が DataGear と等価になる。Code 8 は Agda で書いた CodeGear の型の例である。

Code 8: 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} )
```

GearsOS で CodeGear の性質を証明するには、 Agda で記述された CodeGear と DataGear に対して証明を行う。証明すべき性質は、不定の型を持つ継続 t に記述することができる。例えば、 Code 9 では $(varie) \equiv$ 10 が証明したい命題で、whileTest から、whileLoop の CodeGear に継続した後、この命題が正しければよい。この証明は proof1 の型に対応する λ 項を与えると証明が完了したことになる。ここで与えている refl は左右の項が等しいというもので、ここでの命題が正しいことが証明できている。

Code 9: Agda での証明の例

```
1  proof1 : whileTest 10 λ ( env → whileLoop env λ ( e →
       (vari e) ≡  10 ))
2  proof1 = refl
```

## 7. Agda での HoareLogic

今回は、 Code 10 のような while Loop に対しての検証を行う。

Code 10: while Loop Program

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

Code 10 では最初期の事前条件は何もなく、プログラムが停止するときの条件として、 $i == 10 \wedge n == 0$ が成り立つ。また、 $n = 10$、 $i = 0$、 といった代入に事前条件と、事後条件をつけ、 while 文にループの普遍条件をつけることになる。

現在 Agda 上での HoareLogic は初期の Agda で実装されたもの [3] とそれを現在の Agda に対応させたもの [4] が存在している。

今回は現在の Agda に対応させたもの [4] の Command と証明のためのルールを使って HoareLogic を実装した。Code 11 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 の定義をそのまま使っている。

*Env* は Code 10 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。

Code 11: 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 12 は Code 10 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。

Code 12: 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 13 のような interpreter を記述した。

Code 13: Agda での HoareLogic interpreter

```
1  {−# TERMINATING #−}
2  interpret : Env → Comm → Env
3  interpret env Skip = env
4  interpret env Abort = env
5  interpret env (PComm x) = x env
6  interpret env (Seq comm comm1) = interpret (interpret
       env comm) comm1
7  interpret env (If x then else) with x env
8  ... | true = interpret env then
9  ... | false = interpret env else
10 interpret env (While x comm) with x env
11 ... | true = interpret (interpret env comm) (While x comm)
12 ... | false = env
```

Code 13 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。

Code 14: Agda での HoareLogic の実行

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

Code 14 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。

次に先程書いたプログラムの証明について記述する。

Code 16 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。– これは Pre と Post の Condition を Command で変化させるここでの HTProof [4] も Agda2 に移植されたものを使っている。

PrimRule は Code 15 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 15 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。

Code 15: Axiom と Tautology

```
1  _⇒_ : Bool → Bool → Bool
2  false  ⇒  _ = true
3  true  ⇒  true = true
4  true  ⇒  false = false
5
6  Axiom : Cond → PrimComm → Cond → Set
7  Axiom pre comm post = ∀  (env : Env) →
       (pre env) ⇒ ( post (comm env)) ≡ true
```

```
8
9 Tautology : Cond → Cond → Set
10 Tautology pre post = ∀ (env : Env) →
       (pre env) ⇒ (post env) ≡ true
```

Code 16: Agda での HoareLogic の構成

```
1  data HTProof : Cond -> Comm -> Cond -> Set where
2    PrimRule : {bPre : Cond} -> {pcm : PrimComm} -> {bPost
         : Cond} ->
3              (pr : Axiom bPre pcm bPost) ->
4              HTProof bPre (PComm pcm) bPost
5    SkipRule : (b : Cond) -> HTProof b Skip b
6    AbortRule : (bPre : Cond) -> (bPost : Cond) ->
7               HTProof bPre Abort bPost
8    WeakeningRule : {bPre : Cond} -> {bPre' : Cond} -> {cm
         : Comm} ->
9                {bPost' : Cond} -> {bPost : Cond} ->
10               Tautology bPre bPre' ->
11               HTProof bPre' cm bPost' ->
12               Tautology bPost' bPost ->
13               HTProof bPre cm bPost
14   SeqRule : {bPre : Cond} -> {cm1 : Comm} -> {bMid : Cond
         } ->
15             {cm2 : Comm} -> {bPost : Cond} ->
16             HTProof bPre cm1 bMid ->
17             HTProof bMid cm2 bPost ->
18             HTProof bPre (Seq cm1 cm2) bPost
19   IfRule : {cmThen : Comm} -> {cmElse : Comm} ->
20            {bPre : Cond} -> {bPost : Cond} ->
21            {b : Cond} ->
22            HTProof (bPre /\ b) cmThen bPost ->
23            HTProof (bPre /\ neg b) cmElse bPost ->
24            HTProof bPre (If b cmThen cmElse) bPost
25   WhileRule : {cm : Comm} -> {bInv : Cond} -> {b : Cond}
         ->
26               HTProof (bInv /\ b) cm bInv ->
27               HTProof bInv (While b cm) (bInv /\ neg b)
```

Code 16 を使って Code 10 の whileProgram を証明する。

全体の証明は Code 17 の proof1 の様になる。proof1 では型で initCond、 Code 12 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 の証明になっている。

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 [8] 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。

Code 17: 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 12 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する形になる。

## 8. Gears ベースの HoareLogic の証明

次に Gears をベースにした HoareLogic の例を見ていく。Gears を用いた記述では、 Input の DataGear 、 CodeGear、Output の DataGear という並びでプログラムを記述していく。そのため、Input DataGear を PreCondition、 Command を CodeGear、 Output DataGear を PostCondition とそのまま置き換えることができる。

こちらも通常の HoareLogic と同様に Code 10 の while プログラムと同様のものを記述する Code 18 は、 CodeGear、DataGear を用いた Agda 上での while Program の記述であり、証明でもある。

Code 18: Gears 上での WhileLoop の記述と検証

```
1 proofGears : {c10 : ℕ } → Set
2 proofGears {c10} = whileTest {_} {_} {c10} (λ n p1 →
      conversion1 n p1 (λ n1 p2 →
     whileLoop' n1 p2 (λ n2 →  ( vari n2 ≡ c10 ))))
```

Code 18 で使われている CodeGear を見ていく

始めに Code 19 の whileTest では変数 i、n にそれぞれ 0 と 10 を代入している。whileTest は最初の CodeGear なので initCondition は true で、Code : の後ろに書かれた $(env : Env)- > ((varienv) \equiv 0)/ ((varnenv) \equiv c10)$ が PostCondition である。λ 項の next には次の CodeGear が入り、継続される引数である env は whileTest の PostCondition であり、次の CodeGear の PreCondition にあたる。conversion1 は通常の Condition からループ不変条件に移行するためのもので前の Condition である $i == 0 and n == 10$ が成り立っている時、$i + n == 10$ を返す CodeGear となっている。whileLoop では conversion1 のループ不変条件を受け取って While の条件である $0 < n$ が成り立っている間、$i{+}{+}; n{+}{+}$ を行う。そして、ループを抜けた後の termCondition は $i == 10$ となる。

Code 19: Gears whileProgram

```
1 whileTest : {l : Level} {t : Set l}  → {c10 : ℕ } →
     (Code : (env : Env)  →
2         ((vari env) ≡ 0) /\ ((varn env) ≡ c10) → t) → t
3 whileTest {_} {_} {c10} next = next env proof2
```

```
4    where
5      env : Env
6      env = record {vari = 0 ; varn = c10}
7      proof2 : ((vari env) ≡ 0) /\ ((varn env) ≡ c10)
8      proof2 = record {pi1 = refl ; pi2 = refl}
9
10 conversion1 : {l : Level} {t : Set l } →
       (env : Env) → {c10 : ℕ } →
       ((vari env) ≡ 0) /\ ((varn env) ≡ c10)
11           → (Code : (env1 : Env) → (varn env1 + vari
       env1 ≡ c10) → t) → t
12 conversion1 env {c10} p1 next = next env proof4
13   where
14     proof4 : varn env + vari env ≡ c10
15     proof4 = let open ≡-Reasoning  in
16       begin
17         varn env + vari env
18       ≡⟨ cong ( λ n → n + vari env ) (pi2 p1 ) ⟩
19         c10 + vari env
20       ≡⟨ cong ( λ n → c10 + n ) (pi1 p1 ) ⟩
21         c10 + 0
22       ≡⟨ +-sym {c10} {0} ⟩
23         c10
24       ∎
25
26 {-# TERMINATING #-}
27 whileLoop : {l : Level} {t : Set l} → (env : Env) → {c10 :
       ℕ } →
       ((varn env) + (vari env) ≡ c10) → (Code : Env → t)
       → t
28 whileLoop env proof next with  ( suc zero  ≤? (varn  env)
       )
29 whileLoop env proof next | no p = next env
30 whileLoop env {c10} proof next | yes p = whileLoop env1 (
       proof3 p ) next
31     where
32       env1 = record {varn = (varn  env) - 1 ; vari = (vari
         env) + 1}
33       1<0 : 1 ≤ zero → ⊥
34       1<0 ()
35       proof3 : (suc zero  ≤ (varn  env))  →
         varn env1 + vari env1 ≡ c10
36       proof3 (s≤s lt) with varn  env
37       proof3 (s≤s z≤n) | zero = ⊥-elim (1<0 p)
38       proof3 (s≤s (z≤n {n'}) ) | suc n =  let open ≡-
       Reasoning  in
39         begin
40           n' + (vari env + 1)
41         ≡⟨ cong ( λ z →
         n' + z ) ( +-sym  {vari env} {1} )  ⟩
42           n' + (1 + vari env )
43         ≡⟨ sym ( +-assoc (n')  1 (vari env) ) ⟩
44           (n' + 1) + vari env
45         ≡⟨ cong ( λ z → z + vari env )  +1≡suc  ⟩
46           (suc n' ) + vari env
47         ≡⟨⟩
48           varn env + vari env
49         ≡⟨ proof  ⟩
50           c10
51         ∎
```

## 9. まとめと今後の課題

本研究では Agda 上で HoareLogic の while を使った例題を作成し、証明を行なった。

また、 Gears を用いた HoareLogic を記述することができた。さらに、Gears を用いて HoareLocic 記述で、証明を引数として受け渡して記述することで、証明とプログラムを一体化することができた。

今後の課題としては GearsOS の検証のために、RedBlackTree や SynchronizedQueue などのデータ構造の検証を HoareLogic ベースで行うことなどが挙げられる。

### Referências


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

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

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

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

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

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

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

[8] 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).

[9] 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.

[10] 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.

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

[12] Kaito TOKKMORI and Shinji KONO. Implementing continuation based language in llvm and clang. *LOLA 2015*, July 2015.

[13] 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.

[14] 宮城 光希, 河野 真治. CbC 言語による OS 記述, 2017.

[15] 河野 真治, 伊波 立樹, and 東恩納 琢偉. Code gear、data gear に基づく os のプロトタイプ. 情報処理学会システムソフトウェアとオペレーティング・システム研究会 *(OS)*, May 2016.

[16] 健太 比嘉 and 真治 河野. 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.