修士 (工学) 学位論文

Master's Thesis of Engineering

Continuation based C での Hoare Logic 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 対応にもどつ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。

Abstract

OS and application reliability are important. To increase reliability, verifications of program with specifications are necessary. Floyd-Hoare logic (hereafter Hoare Logic) is a welknown program verification method. Hoare Logic verifies the postonditions of a function are satisfied when the postconditions are satifiles. It also checks the halt condition of the program. Hoare Logic is a useful simple approach but often only applies to a limited set of commands and while programs. It is not generaly suitable for complex ordinary programming languages.

Our laboratory is developping Continuation based C (CbC) as a reliable language. In CbC, programs are described using units of CodeGear and DataGear.

CodeGear can be described in Agda as a function using the description of a light weight continuous passing. Agda is a theorem proof system based Curry Howard correspondence, and it is also a functional programming language. In Agda, conditions can be described as propositions, The continuation can have preconditions and postconditions.

In existing languages, conditions are described in asserts, etc., but the proof cannot be done in that programming language. Since Agda can describe the proof itself, that is, the inference among the propositions, as λ terms, The proof of Hoare Logic itself can be described as Meta CodeGear. This was not possible with existing languages. The point is that the program itself can be described with CodeGear of Agda base. CodeGear has only input and output, and executes continuously in a goto manner without calling a function. This format is naturally define Hoare Logic commands.

Hoare Logic's proof requires three conditions. One, Pre-conditions and post-conditions are connected correctly throughout the program. The preconditions and postconditions are equal in the loop (the connection of CodeGear including the loop) and constitute an invariant condition. In addition, we need to show that the loop stops. For a program that does not stop, it is possible to define partial validity without stopping.

In this paper, we created and proved a simple while Loop program using the description of Hoare Logic on Agda. This proof includes termination and the overall soundness of the proof. Previously, the soundness of Hoare Logic was limited to rather simple commands. However, in this method, individual proofs are given as Meta CodeGears for complex CodeGears.

This made it possible to prove the soundness itself.

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

研究関連論文業績 **i**

第**1**章 プログラミング言語の検証 **4**

第**2**章 **Continuation based C** **5**
2.1 Code Gear と Data Gear 5
2.2 Meta CodeGear、 Meta DataGear 5

第**3**章 定理証明支援系言語 **Agda** **8**
3.1 関数型言語としての Agda 8
3.2 Agda のデータ 9
3.3 Agda の関数 10
3.4 定理証明支援器としての Agda 12

第**4**章 **Hoare Logic** **15**
4.1 Hoare Logic 15
4.2 while program の部分正当性 18
4.3 Hoare Logic での健全性 20

第**5**章 **Continuation based C** と **Agda** **24**
5.1 DataGear、 CodeGear と Agda の対応 24
5.2 Meta Gears の表現 25

第**6**章 **CbC** と **Hoare Logic** **26**
6.1 CbC での while program の記述 26
6.2 CbC での Hoare Logic の記述 27
6.3 CbC 上での Hoare Logic を用いた検証 28

第**7**章 まとめと今後の課題 **32**
7.1 今後の課題 32

謝辞 **32**

付録 **33**

図目次

2.1 CodeGear と DataGear 6
2.2 メタ計算を可視化した CodeGear と DataGear 6

6.1 CodeGear、DataGear での Hoare Logic 28

ソースコード目次

3.1 モジュールのインポートとオプション . . . 8
3.2 自然数を表すデータ型 Nat の定義 . . . 9
3.3 Agda におけるレコード型の定義 . . . 9
3.4 Agda における関数定義 . . . 10
3.5 自然数での加算の定義 . . . 10
3.6 自然数の減算によるパターンマッチの例 . . . 10
3.7 Agda におけるラムダ計算 . . . 11
3.8 Agda における where 句 . . . 11
3.9 停止しない関数 loop、停止する関数 stop . . . 11
3.10 等式変形の例 . . . 12
3.11 rewrite での等式変形の例 . . . 12
3.12 等式変形の例 1/3 . . . 13
3.13 等式変形の例 2/3 . . . 13
3.14 等式変形の例 3/3 . . . 13
4.1 while Loop Program . . . 15
4.2 Agda での Hoare Logic の構成 . . . 16
4.3 while Loop (再掲) . . . 17
4.4 ソースコード **??** と対応した Hoare Logic のプログラム . . . 17
4.5 Agda での Hoare Logic interpreter . . . 17
4.6 Agda での Hoare Logic の実行 . . . 17
4.7 Axiom と Tautology . . . 18
4.8 Agda での Hoare Locig の構成 . . . 18
4.9 while loop の検証用記述 . . . 19
4.10 State Sequence の部分正当性 . . . 20
4.11 Agda での Hoare Logic の健全性 . . . 21
4.12 HTProof の Soundness への適用 . . . 22
4.13 while program の健全性 . . . 22
5.1 Agda での CodeGear の例 . . . 24
5.2 Agda における Meta DataGear . . . 25

5.3 Agda における Meta CodeGear 25
6.1 CbC 上での while program 26
6.2 停止するループ loopP' 27
6.3 CbC 上での Hoare Logic 28
6.4 CbC ベースの Hoare Logic 29
6.5 CbC 上の Hoare Logic での 代入 29
6.6 CbC 上の Hoare Logic での while loop 29
6.7 CbC 上の Hoare Logic 30
6.8 CbC 上での導出中の Soundness 30
6.9 loopPwP' の補助定理 loopHelper 31
6.10 loopHelper を使って導出した Soundness 31

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。実際に kernel 検証を行った例 [?] [?] 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 ATS2[?] や Rust[?] などの低レベル記述向けの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[?]、 Coq[?] 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[?] (CbC) という言語を開発している。

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。

本研究では Agda 上で CodeGear、DataGear という単位を用いてプログラムを記述し、メタ計算部分で Hoare Logic を元にした検証を行った。

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 [?] [?] と gcc [?] [?] 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。

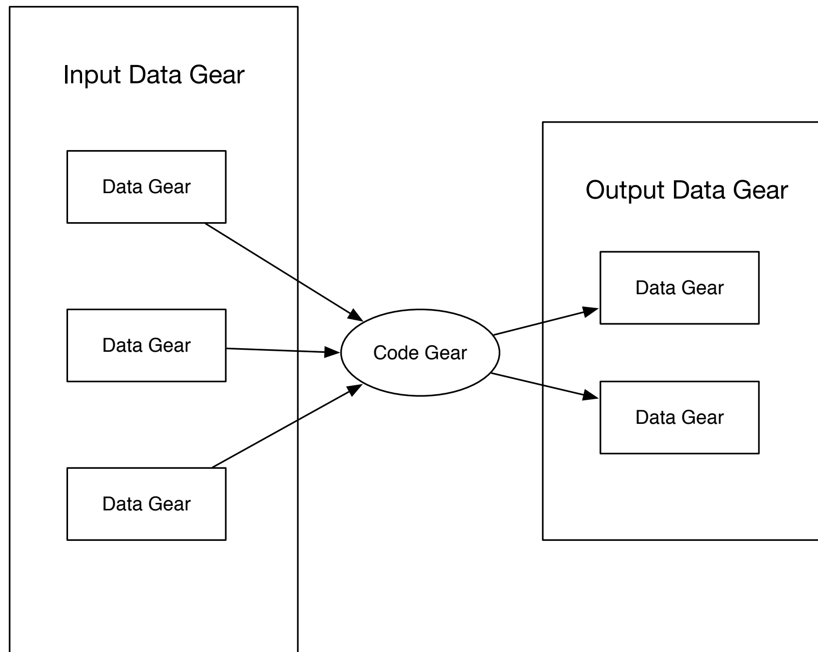

図 2.1: CodeGear と DataGear

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。

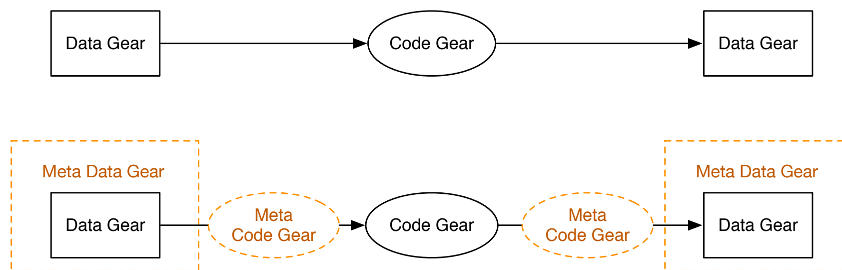

図 2.2: メタ計算を可視化した CodeGear と DataGear

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 [?] 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 [?] 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

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

```
import Level renaming (suc to S) -- import module with rename suc to S
import Data.String hiding (_++_) -- import module without _++_
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。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。

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

```
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。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

```
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。

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。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。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 ℕ 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 : 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ソースコード 3.5 のようになる。

ソースコード 3.5: 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

```
_-_ : 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。ソースコード 3.4 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 : 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` を使うとリストソースコード 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

```
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。ソースコード 3.9 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。

ソースコード 3.9: 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。

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 3.10 を見る。ここでの `+zero` は右から `zero` を足しても $\equiv$ 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 $zero \equiv zero$ 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 `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。

ソースコード 3.10: 等式変形の例

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

-- 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。ソースコード 3.11 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 3.11: rewrite での等式変形の例

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

ソースコード 3.12、ソースコード 3.13、ソースコード 3.14 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 3.12 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 3.12: 等式変形の例 1/3

```
+-comm : (x y : ℕ) → x + y ≡ y + x
+-comm zero y rewrite (+zero {y}) = refl
+-comm (suc x) y = let open ≡-Reasoning in
  begin
  ?0 ≡⟨ ?1 ⟩
  ?2 ∎

-- ?0 : ℕ {(suc x) + y}
-- ?1 : suc x + y ≡ y + suc x
-- ?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 3.13 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 3.13: 等式変形の例 2/3

```
+-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) ≡⟨ ?0 ⟩
  ?1 ∎

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

ソースコード 3.14 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 3.14: 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 [?](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。

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 コマンド 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[?] で実装されたものとそれを現在の Agda に対応させたもの [?] が存在している。

ここでは現在 Agda に対応した Hoare Logic を使用する。

例として ソースコード 4.1 のようなプログラムを記述した。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。

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。

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

ソースコード 4.1: while Loop Program

```
n = 10;
i = 0;

while (n > 0) {
```

```
  i++;
  n--;
}
```

ソースコード 4.2 は Agda 上での Hoare Logic の構築子である。Env は ソースコード 4.1 の n、i といった変数をレコード型でまとめたもので、n と i それぞれが型として Agda 上での自然数の型である ℕ 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 条件で、 Env を受け取って Bool 値、true か false 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 コマンド を表す。

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

PComm は PrimComm を受けて コマンド 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 コマンド を 2 つ受けて コマンド を返す型で定義されている。これは、ある コマンド から コマンド に移り、その結果を次の コマンド 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 コマンド 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 コマンド である。

ソースコード 4.2: Agda での Hoare Logic の構成

```
PrimComm : Set
PrimComm = Env → Env

Cond : Set
Cond = (Env → Bool)

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 上の Hoare Logic で使われるプログラムは Comm 型の関数となる。プログラムはコマンド Comm 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。

ソースコード 4.4 は ソースコード 4.1 で書いた While Loop を Hoare Logic でのコマンド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。

比較しやすいように ソースコード 4.1 を ソースコード 4.3 に再掲した。

ソースコード 4.3: while Loop (再掲)

```
n = 10;
i = 0;

while (n > 0) {
  i++;
  n--;
}
```

ソースコード 4.4: ソースコード 4.3 と対応した Hoare Logic のプログラム

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

この Comm を Agda 上で実行するため、 ソースコード 4.5 の `interpret` 関数を作成した。

ソースコード 4.5: 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
```

ソースコード 4.5 は 初期状態の `Env` と 実行する コマンド 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。`interpret` 関数は停止性を考慮していないため、`{-# TERMINATING #-}` タグを付けている。

ソースコード 4.6 のように `interpret` に `vari = 0 , varn = 0` の `record` を渡し、実行する `Comm` を渡して 評価すると `record { varn = 0 ; vari = 10 }` のような `Env` が返ってくる。`interpret` で実行される コマンド は ソースコード 4.4 で記述した While Loop するコマンドである

ソースコード 4.6: Agda での Hoare Logic の実行

```
test : Env
```

```
test =  interpret ( record { vari = 0  ; varn = 0 } ) program
-- record { varn = 0 ; vari = 10 }
```

4.2　while program の部分正当性

ここでは先程記述した ソースコード 4.4 の検証を行う。Hoare Logic ではコマンドに対応した仕様が存在しており、それらを組み合わせた形で仕様を記述する必要がある。この仕様を記述する際、部分正当性が成り立っている必要がある。部分正当性とは Hoare Logic のコマンドが実行される前には事前条件が成り立っていて、コマンドが停止したとき事後条件が成り立っていることである。

ソースコード 4.8 の `HTProof` は Agda 上での Hoare Logic でのコマンドに対応した性質を型としてまとめたものである。`HTProof` では Pre-Condition とコマンド、Post-Condition を受け取って定義される Agda のデータである。ソースコード 4.2 のコマンドで定義された `Skip`、 `Abort`、 `PComm`、`Seq`、`If`、`While`、に対応した証明のための命題が存在している。

`PrimRule` は Pre-Condition と PrimComm、Post-Condition、ソースコード 4.7 の `Axiom` を引数として `PComm` の入った `HTProof` を返す。

`SkipRule` は Condition を受け取ってそのままの Condition を返す HTProof を返す。

`AbortRule` は Pre-Contition を受け取って、`Abort` を実行する HTProof を返す。

`WeakeningRule` は通常の Condition から制約を緩める際にに使用される。4.7 の `Tautology` を使って Condition が同じであることを

`SeqRule` は 3 つの Condition と 2 つの コマンド 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 コマンド 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 コマンド を変えるルールである。この時、どちらかの コマンド 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 コマンド と 2 つの Condition を受け取り、事前条件が成り立っている間、 コマンド を繰り返すことを保証している。

ソースコード 4.7: 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 4.8 を使って ソースコード 4.1 の while program の仕様を構成する。

ソースコード 4.8: 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 4.9 の `proof1` の様になる。`proof1` では型で initCond、 Code 4.4 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 の中にそれぞれの部分正当性を検証するための証明存在しており、それぞれ `lemma` で埋められている。`lemma1` から `lemma5` の証明は概要のみを示し、全体は付録に載せることにする。

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 4.9: while loop の検証用記述

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

Hoare Logic ではコマンドに対応した仕様が存在するため、`proof1` はソースコード 4.4 の program に近い記述になる。

4.3 Hoare Logic での健全性

ソースコード 4.9 では Agda での Hoare Logic を用いた仕様の構成を行った。ここでは、ソースコード 4.9 で構成した仕様が実際に正しく動作するかどうか (健全性) の検証を行う。

ソースコード 4.10 の `SemComm` では各 Comm で成り立つ関係を返す。`Satisfies` では 事前条件 と コマンド、 事後条件 を受け取って、これらが、正しく Comm で成り立つ関係を構築する。

ソースコード 4.10: State Sequence の部分正当性

```
SemComm : Comm → Rel State (Level.zero)
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
```

実行する際、これらの関係を満たしていることで健全性が証明できる。

ソースコード 4.11 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。Soundness のコードは量が多いため部分的に省略し、全文は付録に載せることにする。

ソースコード 4.11: 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} (proj_2 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
    in tautValid bPost' bPost tautPost s2 (hyp s1 s2 (tautValid bPre bPre
    ' tautPre s1 q1) q2)
Soundness (SeqRule {bPre} {cm1} {bMid} {cm2} {bPost} pr1 pr2)
           s1 s2 q1 q2
  = let hyp1 : Satisfies bPre cm1 bMid
        hyp1 = Soundness pr1
        hyp2 : Satisfies bMid cm2 bPost
        hyp2 = Soundness pr2
    in hyp2 (proj_1 q2) s2 (hyp1 s1 (proj_1 q2) q1 (proj_1 (proj_2 q2))) (
    proj_2 (proj_2 q2))
Soundness (IfRule {cmThen} {cmElse} {bPre} {bPost} {b} pThen pElse)
          s1 s2 q1 q2
  = let hypThen : Satisfies (bPre ∧ b) cmThen bPost
        hypThen = Soundness pThen
        hypElse : Satisfies (bPre ∧ neg b) cmElse bPost
        hypElse = Soundness pElse
        rThen : RelOpState.comp (RelOpState.delta (SemCond b))
                  (SemComm cmThen) s1 s2 → SemCond bPost s2
        rThen = λ h → hypThen s1 s2 ((proj_2 (respAnd bPre b s1)) (q1 ,
    proj_1 t1))
          (proj_2 ((proj_2 (RelOpState.deltaRestPre (SemCond b) (SemComm
    cmThen) s1 s2)) h))
        rElse : RelOpState.comp (RelOpState.delta (NotP (SemCond b)))
                  (SemComm cmElse) s1 s2 → SemCond bPost s2
        rElse = λ h →
                  let t10 : (NotP (SemCond b) s1) × (SemComm cmElse s1 s2
    )
```

```
                      t10 = proj_2 (RelOpState.deltaRestPre
                                  (NotP (SemCond b)) (SemComm cmElse) s1
   s2) h
                 in hypElse s1 s2 (proj_2 (respAnd bPre (neg b) s1)
                           (q1 , (proj_2 (respNeg b s1) (proj_1 t10))))
   (proj_2 t10)
    in when rThen rElse q2
Soundness (WhileRule {cm'} {bInv} {b} pr) s1 s2 q1 q2
  = proj_2 (respAnd bInv (neg b) s2)
          (lem1 (proj_1 q2) s2 (proj_1 t15) , proj_2 (respNeg b s2) (proj
   _2 t15))
    where
      hyp : Satisfies (bInv ∧ b) cm' bInv
      hyp = Soundness pr
      Rel1 : ℕ → Rel State (Level.zero)
      Rel1 = λ m →
               RelOpState.repeat
                 m
                 (RelOpState.comp (RelOpState.delta (SemCond b))
                                  (SemComm cm'))
      t15 : (Rel1 (proj_1 q2) s1 s2) × (NotP (SemCond b) s2)
      t15 = proj_2 (RelOpState.deltaRestPost
        (NotP (SemCond b)) (Rel1 (proj_1 q2)) s1 s2) (proj_2 q2)
      lem1 : (m : ℕ) → (ss2 : State) → Rel1 m s1 ss2 → SemCond bInv
   ss2
      lem1 zero ss2 h = substId1 State (proj_2 h) (SemCond bInv) q1
      lem1 (suc n) ss2 h
    = let hyp2 : (z : State) → Rel1 (proj_1 q2) s1 z →
                     SemCond bInv z
              hyp2 = lem1 n
              t22 : (SemCond b (proj_1 h)) × (SemComm cm' (proj_1 h) ss2)
              t22 = proj_2 (RelOpState.deltaRestPre (SemCond b) (SemComm
   cm') (proj_1 h) ss2)
                    (proj_2 (proj_2 h))
              t23 : SemCond (bInv ∧ b) (proj_1 h)
              t23 = proj_2 (respAnd bInv b (proj_1 h))
                (hyp2 (proj_1 h) (proj_1 (proj_2 h)) , proj_1 t22)
      in hyp (proj_1 h) ss2 t23 (proj_2 t22)
```

ソースコード 4.12 は HTProof で記述された仕様を、実際に満たすことが可能であることを Satisfies が返す。証明部分では 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 4.12: HTProof の Soundness への適用

```
PrimSoundness : {bPre : Cond} -> {cm : Comm} -> {bPost : Cond} ->
            HTProof bPre cm bPost -> Satisfies bPre cm bPost
PrimSoundness {bPre} {cm} {bPost} ht = Soundness ht
```

ソースコード 4.13 では ソースコード 4.4 の program の Hoare Logic での命題である。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 4.13: while program の健全性

```
proofOfProgram : (c10 : $mathbb{N}$) → (input output : Env )
  → initCond input ≡ true
  → (SemComm (program c10) input output)
  → termCond {c10} output ≡ true
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 `proof1` を \verbPrimSoundness/ に入力として渡す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。

ソースコード 5.1 は Agda で記述した加算を行う CodeGear の例である。

ソースコード 5.1: 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
--    λ next → next 30
```

`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。

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単位である。Meta DataGear はメタ計算で使われる DataGear で、実行するメタ計算によって異なる。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られる。Agda 上では Meta DataGear を持つことでデータ構造自体が関係を持つデータを作ることができる。ソースコード 5.2 は While Program での制約条件をまとめたものである。

ソースコード 5.2: Agda における Meta DataGear

```
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)
```

ここでは `whileTestState` で Meta DataGear を識別するためのデータを分け、`whileTestStateP` でそれぞれの Meta DataGear を返している。ここでは `(vari env ≡ 0)` (varn env ≡ c10 env)/ などのデータを Meta DataGear として扱う。

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 Meta DataGear を引数に取りそれらの関係を返す CodeGear である。

メタ計算で検証を行う際の Meta CodeGear は Agda で記述した CodeGear の検証そのものである。例として ソースコード 5.3 を示す。

ソースコード 5.3: Agda における Meta CodeGear

```
whileTestPwP : {l : Level} {t : Set l} → (c10 : ℕ) →
             ((env : Envc ) → (mdg : (vari env ≡ 0) ∧ (varn env ≡ c10
   env)) → t) → t
whileTestPwP c10 next = next env record { pi1 = refl ; pi2 = refl } where
  env : Envc
  env = whileTestP c10 ( λ env → env )
```

`whileTestPwP` は Meta CodeGear の例である。ここでは Meta DataGear に `mdg` という名前をつけてある。この Meta CodeGear では次の CodeGear に mdg を渡しており、CodeGear 内で Meta DataGear の性質が正しいことを検証して次の CodeGear に遷移することがわかる。

Meta CodeGear はこのような形で記述される。

第6章　CbC と Hoare Logic

第 4 章 では Agda 上での Hoare Logic を用いて検証を行った。第 5 章 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。

本章では 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 での while program の記述

検証を始める前に CbC 上で while program を実装する。

6.1 は CbC 上での while progam に相当する CodeGear である。ここでは 変数 i、 n、入力として受ける c10 を record 型でまとめて Envc としている。

ソースコード 6.1: CbC 上での while program

```
whileTestP : {l : Level} {t : Set l} → (c10 : ℕ) → (Code : Envc → t)
    → t
whileTestP c10 next = next (record {c10 = c10 ; varn = c10 ; vari = 0 })

whileLoopP’ : {l : Level} {t : Set l} → Envc → (next : Envc → t) → (
    exit : Envc → t) → t
whileLoopP’ record { c10 = c10 ; varn = zero ; vari = vari } _ exit =
    exit record { c10 = c10 ; varn = zero ; vari = vari }
whileLoopP’ record { c10 = c10 ; varn = suc varn1 ; vari = vari } next _
    = next (record {c10 = c10 ; varn = varn1 ; vari = suc vari })

{-# TERMINATING #-}
loopP : {l : Level} {t : Set l} → Envc → (exit : Envc → t) → t
loopP env exit = whileLoopP’ env (λ env → loopP env exit ) exit

whileTestPCall : (c10 :  ℕ ) → Envc
whileTestPCall c10 = whileTestP {_} {_} c10 (λ env → loopP env (λ env
    →  env))

-- whileTestPCall 10
-- record { c10 = 10 ; varn = 0 ; vari = 10 }
```

whileTestP は代入を行う CodeGear で、継続に構築した Env を渡している。

ループ部分はループの判断をする CodeGear と ループを行う CodeGear の 2 つに分けて記述している。whileLoopP' はループを判別する CodeGear で、varn の値が zero かそうでないかでループを終えるか続けるかを判断している。loopP はループを行う CodeGear で　継続先に whileLoopP と更に whileLoopP の継続先に自身である loopP を記述することでループさせている。このままでは停止しないため {-# TERMINATING #-} をつけている。

停止する記述は、loopP' として ソースコード 6.2 のように記述するとよい。

ソースコード 6.2: 停止するループ loopP'

```
loopP' : {l : Level} {t : Set l} → Envc → (exit : Envc → t) → t
loopP' record { c10 = c10 ; varn = zero ; vari = vari } exit =
  exit (record { c10 = c10 ; varn = zero ; vari = vari })
loopP' record { c10 = c10 ; varn = (suc varn_1) ; vari = vari } exit =
  whileLoopP' (record { c10 = c10 ; varn = (suc varn_1) ; vari = vari })
  (λ env → loopP' (record { c10 = c10 ; varn = varn_1 ; vari = vari })
   exit ) exit

whileTestPCall' : (c10 :  ℕ ) → Envc
whileTestPCall' c10 = whileTestP' {_} {_} c10 (λ env → loopP' env (λ env
    →  env))

-- whileTestP' 10
-- record { c10 = 10 ; varn = 0 ; vari = 10 }
```

whileTestPCall は実際に CodeGear を組み合わせたプログラムである。while program の n の値 10 を入れて whileTestPCall 10 のように実行すると、record c10 = 10 ; varn = 0 ; vari = 10 が帰ってくる。loopP' でも同様の実行ができる。

6.2 CbC での Hoare Logic の記述

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。

CbC での Hoare Logic は 図 6.1 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。

6.3 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。

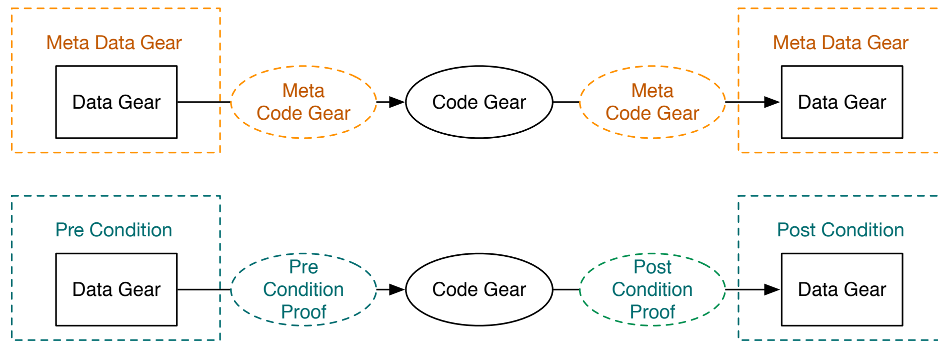

図 6.1: CodeGear、DataGear での Hoare Logic

ソースコード 6.3: 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。

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。

6.3 CbC 上での Hoare Logic を用いた検証

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。

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。

ソースコード 6.4 は while program の 3 つの状態を記述したものである。

ソースコード 6.4: 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。

ソースコード 6.5 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。

ソースコード 6.5: 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 )
```

ソースコード 6.6 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。

ソースコード 6.6: 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 を使い仕様を記述し、検証する。

ソースコード 6.7: 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'` はソースコード 6.5 やソースコード 6.6 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。

`whileTestPCallwP'` を検証するには、ソースコード 6.8 のように ℕ を受け取って `whileTestPCallwP' ℕ` が成り立つ型を記述し、実際に導出部分で定義する必要がある。

ソースコード 6.8: CbC 上での導出中の Soundness

```
whileCallwP : (c : ℕ) → whileTestPCallwP' c
whileCallwP c = whileTestPwP {_} {_} c
  (λ env s → loopPwP' (c10 env) env (sym (pi2 s)) (conv env s) {!!})


-- Goal: (env_1 : Envc) →
--        vari env_1 ≡ c10 env_1 →
--        loopPwP' c (whileTestP c (λ env_2 → env_2)) refl +zero
--        (λ env_2 s_1 → vari env_2 ≡ c10 env_2)
```

```
-- ————————————————————————————————
-- s   : (vari env ≡ 0) ∧ (varn env ≡ c10 env)
-- env : Envc
-- c   : ℕ
```

whileTestPwP 代入のため単純に Agda が計算できる。しかし、 loopPwP' はソースコード 6.8 のコメントのように実際の値が入るまで計算をすすめることができなかった。そのためソースコード 6.10 で、 loop を簡約する補助定理 loopHelper を記述した。

ソースコード 6.9: loopPwP' の補助定理 loopHelper

```
loopHelper : (n : ℕ) → (env : Envc ) → (eq : varn env ≡ n) → (seq :
   whileTestStateP s2 env)
          → loopPwP' n env (sym eq) seq (λ env_1 x → (vari env_1 ≡ c10
    env_1))
loopHelper zero env eq refl rewrite eq = refl
loopHelper (suc n) env refl refl =
          loopHelper n (record { c10 = suc (n + vari env) ; varn = n ;
   vari = suc (vari env) }) refl (+-suc n (vari env))
```

loopHelper では loopPwP' が必ず停止し、vari env ≡ c10 env が成り立つことを証明している。

ソースコード 6.10: loopHelper を使って導出した Soundness

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

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　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。

7.1　今後の課題

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。

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: 2020/2/9(Sun).

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

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

[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] 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).

[10] 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.

[11] 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).

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

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

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

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

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

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

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

[25] 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).

[26] 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.

[27] 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.