

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ミング言語でもある。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 wellknown program verification method. Hoare Logic verifies the postconditions of a function are satisfied when the preconditions are satisfied. 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 generally suitable for complex ordinary programming languages.

Our laboratory is developing 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	6
第 2 章 Continuation based C	7
2.1 Code Gear と Data Gear	7
2.2 Meta CodeGear、Meta DataGear	7
第 3 章 定理証明支援系言語 Agda	10
3.1 関数型言語としての Agda	10
3.2 Agda のデータ	11
3.3 Agda の関数	12
3.4 定理証明支援器としての Agda	14
第 4 章 Hoare Logic	17
4.1 Hoare Logic	17
4.2 while program の仕様記述	20
4.3 Hoare Logic の健全性	22
第 5 章 Continuation based C と Agda	26
5.1 DataGear、CodeGear と Agda の対応	26
5.2 Meta Gears の表現	27
第 6 章 CbC と Hoare Logic	28
6.1 CbC での while program の記述	28
6.2 CbC での Hoare Logic の記述	29
6.3 CbC 上での Hoare Logic を用いた仕様記述	30
6.4 CbC 上での Hoare Logic を用いた検証の健全性	33
第 7 章 まとめと今後の課題	37
7.1 今後の課題	37

謝辞	38
参考文献	40
付録	42
付録 A ソースコード一覧	43
A-1 Agda 上の Hoare Logic のコマンドで記述された while program (whileTest-Prim.agda)	43
A-2 Hoare Logic での while program の検証 (whileTestPrimProof.agda)	45
A-3 CbC の Hoare Logic をベースにした while program の仕様記述と検証 (whileTestGears.agda)	51

目 次

2.1	CodeGear と DataGear	8
2.2	メタ計算を可視化した CodeGear と DataGear	8
6.1	CodeGear、DataGear での Hoare Logic	30

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	10
3.2	自然数を表すデータ型 Nat の定義	11
3.3	Agda におけるレコード型の定義	11
3.4	Agda における関数定義	12
3.5	自然数での加算の定義	12
3.6	自然数の減算によるパターンマッチの例	13
3.7	Agda におけるラムダ計算	13
3.8	Agda における where 句	13
3.9	停止しない関数 loop、停止する関数 stop	14
3.10	等式変形の例	14
3.11	rewrite での等式変形の例	15
3.12	等式変形の例 1/3	15
3.13	等式変形の例 2/3	15
3.14	等式変形の例 3/3	16
4.1	while Loop Program	17
4.2	Agda での Hoare Logic の構成	18
4.3	while Loop (再掲)	19
4.4	ソースコード 4.3 と対応した Hoare Logic のプログラム	19
4.5	Agda での Hoare Logic interpreter	19
4.6	Agda での Hoare Logic の実行	20
4.7	Axiom と Tautology	21
4.8	Agda での Hoare Logic の構成	21
4.9	while loop の検証用記述	22
4.10	State Sequence の部分正当性	22
4.11	Agda での Hoare Logic の健全性	23
4.12	HTPProof の Soundness への適用	25
4.13	while program の健全性	25
5.1	Agda での CodeGear の例	26
5.2	Agda における Meta DataGear	27

5.3	Agda における Meta CodeGear	27
6.1	CbC 上での while program	28
6.2	停止するループ loopP'	29
6.3	CbC 上での Hoare Logic	30
6.4	CbC ベースの Hoare Logic	31
6.5	CbC 上の Hoare Logic での 代入	31
6.6	CbC 上の Hoare Logic での while loop	31
6.7	CbC 上の Hoare Logic	32
6.8	接続途中の CbC 上での CodeGear の条件接続	32
6.9	loopPwP' の補助定理 loopHelper	33
6.10	完成した CbC 上での CodeGear の条件接続	33
6.11	implies	34
6.12	代入を行う whileTestP の健全性	34
6.13	条件を変更する Conversion の健全性	34
6.14	while loop をする loopPP の健全性	34
6.15	loopPP を証明するための補助定理 loopPPSem と whileLoopPSem	35
A.1	Agda 上での Hoare Logic を用いた while program と検証	43
A.2	Agda 上での Hoare Logic を用いた while program の検証	45
A.3	Agda 上での Hoare Logic を用いた while program と検証	51

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 [1] [2] 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[3] や Rust[4] 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。

定理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] (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 CodeGear 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[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] [9] と gcc [10] [11] 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。

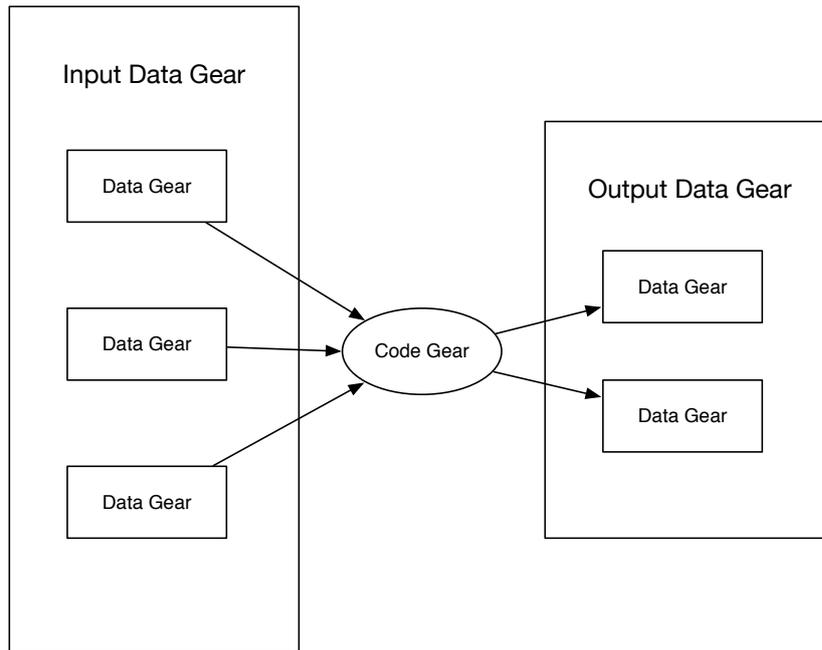


図 2.1: CodeGear と DataGear

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。

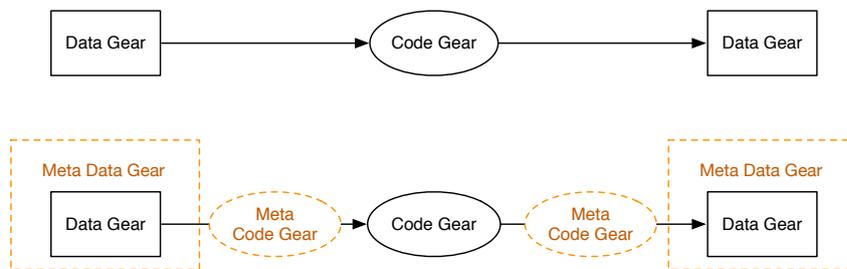


図 2.2: メタ計算を可視化した CodeGear と DataGear

例として CodeGear が DataGear から値を取得する際に使われる、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 として 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 を定義している。

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对一で対応するため 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] は純粋な関数型の言語である。

Agda では C 言語や Python 言語などの手続き型言語との違いがいくつか存在する。そのうちのひとつとして、Agda では変数への再代入が許されていない事が挙げられる。また、for 文や if 文といった構文は用意されておらず、前者は再帰呼び出し、後者はパターンマッチと呼ばれる構文で記述されることが多い。

Agda では型が明示される必要があり、データのみではなく関数にも型を記述する必要がある。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

```

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。

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 \mathbb{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 \mathbb{N} を受け取って \mathbb{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。

\mathbb{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 `vari` と `varn`、`c10` の 3 つの自然数からなるレコード `Envc` を定義するとソースコード 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 Envc : Set where
2 |   field
3 |     vari : N
4 |     varn : N
5 |     c10  : N
6 |

```

```

7 | makeEnv : N → N → N → Env
8 | makeEnv i n c = record { vari = i ; varn = n ; c10 = c }

```

レコードを構築する際は ソースコード 3.3 の `makeEnv` のように `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ある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` で戻り値が型 `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 | +1 : N → N
2 | +1 m = suc m

```

この `+1` に `+1 zero` のように値を与えて評価すると、`suc zero` が返ってくる。

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

```

1 | _+_ : N → N → N
2 | zero + m = m
3 | 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。

パターンマッチでは複数のパターンをまとめて受けることもできる。

例えばソースコード 3.6 の減算では初めのパターンで `n` として最初の引数を受け取っている。この場合は先頭の引数である `n` に入りうる `zero` または `suc n` の 2 つのパターンを同時に受け、2 つ目の引数が `zero` であればそのまま `n` を返している。

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
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。ここでの 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 `λ+1` ように書くことができる。`+1` と `λ+1` は同様の動作をする。

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

```

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)

```

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 \mathbb{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

```

1 {-# TERMINATING #-}
2 loop : ℕ → ℕ
3 loop n = loop (pred n)
4
5 -- pred : ℕ → ℕ
6 -- pred zero    = zero
7 -- pred (suc n) = n
8
9 stop : ℕ → ℕ
10 stop zero = zero
11 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。これは Curry-Howard 対応 [12] に基づくものである。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

```

1 +zero : { y : ℕ } → y + zero ≡ y
2 +zero {zero} = refl
3 +zero {suc y} = cong suc ( +zero {y} )
4
5 -- cong : ∀ (f : A → B) {x y} → x ≡ y → f x ≡ f y
6 -- 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 `\equiv -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

```

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 3.12、ソースコード 3.13、ソースコード 3.14 は \equiv -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 \equiv -Reasoning in begin` と記述し、変形前 \equiv (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

```

1 +-comm : (x y : ℕ) → x + y ≡ y + x
2 +-comm zero y rewrite (+zero {y}) = refl
3 +-comm (suc x) y = let open  $\equiv$ -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 $(\text{suc } x) + y$ を $\text{suc } (x + y)$ 変形してやり、 $y + (\text{suc } x)$ も同様に $\text{suc } (x + y)$ 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 $(\text{suc } x) + y$ は $\text{suc } (x + y)$ に変換できる。ソースコード 3.13 では $\text{suc } (x + y)$ 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 $\text{suc } (y + x)$ へ変換している。

ソースコード 3.13: 等式変形の例 2/3

```

1 +-comm : (x y : ℕ) → x + y ≡ y + x
2 +-comm zero y rewrite (+zero {y}) = refl
3 +-comm (suc x) y = let open  $\equiv$ -Reasoning in
4   begin
5     (suc x) + y ≡⟨ ⟩
6     suc (x + y) ≡⟨ cong suc (+comm x y) ⟩
7     suc (y + x) ≡⟨ ?0 ⟩
8     ?1 ■
9
10 -- ?0 : suc (y + x) ≡ y + suc x
11 -- ?1 : y + suc x

```

ソースコード 3.14 では $\text{suc } (y + x) \text{ equiv } y + (\text{suc } x)$ 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 $y + (\text{suc } x) \text{ equiv } \text{suc } (y + x)$ の形にし、 $y + (\text{suc } x)$ が $\text{suc } (y + x)$ 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

```

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     suc (x + y) ≡⟨
6     suc (x + y) ≡⟨ cong suc (+-comm x y) ⟩
7     suc (y + x) ≡⟨ sym (+-suc {y} {x}) ⟩
8     y + suc x ■
9
10 -- +-suc : {x y : ℕ} → x + suc y ≡ suc (x + y)
11 -- +-suc {zero} {y} = refl
12 -- +-suc {suc x} {y} = cong suc (+-suc {x} {y})

```

Agda ではこのように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 [13](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 、 コマンドを C 、 事後条件を Q としたとき、

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

といった形で表される。この3つ組は Hoare Triple と呼ばれる。

Hoare Triple の事後条件を受け取り異なる条件を返す別の Hoare Triple を繋げることでプログラムを記述していく。

Hoare Logic の検証では、「条件がすべて正しく接続されている」かつ「コマンドが停止する」ことが必要である。これらを満たし、事前条件から事後条件を導けることを検証することで Hoare Logic の健全性を示すことができる。

本章は Agda で実装された Hoare Logic について解説し、Hoare Logic を用いた検証と健全性について説明する。

4.1 Hoare Logic

現在 Agda 上での Hoare Logic は初期の Agda [14] で実装されたものとそれを現在の Agda に対応させたもの [15] 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。

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

```
1 | n = 10;
```

```

2 | i = 0;
3 |
4 | while (n > 0) {
5 |     i++;
6 |     n--;
7 | }

```

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

```

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)

```

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

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

```

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 program は Comm 型のデータで作られた構文木と考えられる。

この構文木を解釈し、Agda 上で実行するには、ソースコード 4.5 の interpret 関数を用いる。

ソースコード 4.5: Agda での Hoare Logic 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
```

interpret 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 を返す関数である。この関数に初期の環境である Env と実行するコマンド Comm を渡すとコマンドに対応した処理を行い、実行後の環境が Env として返ってくる。Agda には Termination Checker[16] が付いており、停止性が明確でないとエラーを出す。interpret は明確に停止する条件がないため、Agda でのコンパイル時にエラーが出る。

今回は停止条件を考えないために `{-# TERMINATING #-}` タグを付けることでエラーを抑制している。

ソースコード 4.6 のように `interpret` に `vari = 0` , `varn = 10` の `record` を渡し、ソースコード 4.4 で記述した `program` を代入した。

ソースコード 4.6: Agda での Hoare Logic の実行

```
1 test : Env
2 test = interpret ( record { vari = 0 ; varn = 10 } ) program
```

実際にこの `test` 関数の評価を行うと `record {vari = 10 ; varn = 0 }` の `Env` が値として返る。これにより、`program` として記述したコマンドベースのプログラムが Agda 上で実行できていることが分かる。

4.2 while program の仕様記述

先程記述したソースコード 4.4 の `program` ではコマンドベースの実行のみで事前、事後条件が存在していなかった。ここではコマンドの事前、事後条件を接続した形で `program` を記述していく。事前条件や事後条件が接続された 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、事前の条件から必ず事後の条件が導かれている必要がある。

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` では事前条件とコマンド、事後条件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` は事前条件と `PrimComm`、事後条件、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

```

1   $\Rightarrow$  : Bool  $\rightarrow$  Bool  $\rightarrow$  Bool
2  false  $\Rightarrow$  _ = true
3  true  $\Rightarrow$  true = true
4  true  $\Rightarrow$  false = false
5
6  Axiom : Cond  $\rightarrow$  PrimComm  $\rightarrow$  Cond  $\rightarrow$  Set
7  Axiom pre comm post =  $\forall$  (env : Env)  $\rightarrow$  (pre env)  $\Rightarrow$  (post (comm env))
    $\equiv$  true
8
9  Tautology : Cond  $\rightarrow$  Cond  $\rightarrow$  Set
10 Tautology pre post =  $\forall$  (env : Env)  $\rightarrow$  (pre env)  $\Rightarrow$  (post env)  $\equiv$  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 Logic の構成

```

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

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、全体は付録ソースコード A-2 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 事前条件が成り立つとき代入を行うと代入後の事後条件が成り立つ証明、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

```

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)}
6       lemma2 (
7         WhileRule {_} {λ e → Equal ((varn e) + (vari e)) 10}
8           $ SeqRule (PrimRule {λ e → whileInv e ∧ lt zero (varn e) }
9             lemma3 )
10          $ 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 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 で構成した仕様が健全であるか検証する。

ここでの健全性はコマンドの仕様である HTProof がすべてつながっていて、事前条件が成り立っているとき、コマンドが停止すると事後条件が成り立つことと、コマンドは条件を意図した通りに遷移させることができることの両方を検証することで、その両方が検証できたとき、この実行は正しいというものである。

ソースコード 4.10 の SemComm では各 Comm で成り立つ関係を返す。Hoare Logic のすべてのコマンドの事前条件から事後条件への遷移が記述されており、Satisfies では事前条件とコマンド、事後条件を受け取って、Comm がでの実行が SemComm を満たすことを示している。

詳しい記述は、現在の Agda に直した Hoare Logic の定義 [15] の RelOp.agda に記述されている。

ソースコード 4.10: State Sequence の部分正当性

```

1 SemComm : Comm → Rel State (Level.zero)
2 SemComm Skip = RelOpState.deltaGlob
3 SemComm Abort = RelOpState.emptyRel
4 SemComm (PComm pc) = PrimSemComm pc
5 SemComm (Seq c1 c2) = RelOpState.comp (SemComm c1) (SemComm c2)
6 SemComm (If b c1 c2)
7   = RelOpState.union
8     (RelOpState.comp (RelOpState.delta (SemCond b))
9                      (SemComm c1))
10    (RelOpState.comp (RelOpState.delta (NotP (SemCond b)))
11                     (SemComm c2))
12 SemComm (While b c)
13   = RelOpState.unionInf
14     (λ (n :  $\mathbb{N}$ ) →
15       RelOpState.comp (RelOpState.repeat
16         n
17         (RelOpState.comp
18           (RelOpState.delta (SemCond b))
19           (SemComm c)))
20       (RelOpState.delta (NotP (SemCond b))))
21
22 Satisfies : Cond → Comm → Cond → Set
23 Satisfies bPre cm bPost
24   = (s1 : State) → (s2 : State) →
25     SemCond bPre s1 → SemComm cm s1 s2 → SemCond bPost s2

```

コマンドを実行する際、これらの関係を満たしていることでコマンドの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、全文は現在の Agda に直した Hoare Logic の定義 [15] の Soundness.agda を確認してほしい。

ソースコード 4.11: Agda での Hoare Logic の健全性

```

1 Soundness : {bPre : Cond} → {cm : Comm} → {bPost : Cond} →
2   HTProof bPre cm bPost → Satisfies bPre cm bPost
3 Soundness (PrimRule {bPre} {cm} {bPost} pr) s1 s2 q1 q2
4   = axiomValid bPre cm bPost pr s1 s2 q1 q2
5 Soundness {.bPost} {.Skip} {bPost} (SkipRule .bPost) s1 s2 q1 q2
6   = substId1 State {Level.zero} {State} {s1} {s2} (proj_2 q2) (SemCond
7     bPost) q1
8 Soundness {bPre} {.Abort} {bPost} (AbortRule .bPre .bPost) s1 s2 q1 ()
9 Soundness (WeakeningRule {bPre} {bPre'} {cm} {bPost'} {bPost} tautPre pr
10   tautPost)
11   s1 s2 q1 q2
12   = let hyp : Satisfies bPre' cm bPost'
13     hyp = Soundness pr
14     in tautValid bPost' bPost tautPost s2 (hyp s1 s2 (tautValid bPre bPre
15       ' tautPre s1 q1) q2)
16 Soundness (SeqRule {bPre} {cm1} {bMid} {cm2} {bPost} pr1 pr2)
17   s1 s2 q1 q2

```

```

15 = let hyp1 : Satisfies bPre cm1 bMid
16     hyp1 = Soundness pr1
17     hyp2 : Satisfies bMid cm2 bPost
18     hyp2 = Soundness pr2
19     in hyp2 (proj_1 q2) s2 (hyp1 s1 (proj_1 q2) q1 (proj_1 (proj_2 q2))) (
20     proj_2 (proj_2 q2))
21 Soundness (IfRule {cmThen} {cmElse} {bPre} {bPost} {b} pThen pElse)
22     s1 s2 q1 q2
23 = let hypThen : Satisfies (bPre ∧ b) cmThen bPost
24     hypThen = Soundness pThen
25     hypElse : Satisfies (bPre ∧ neg b) cmElse bPost
26     hypElse = Soundness pElse
27     rThen : RelOpState.comp (RelOpState.delta (SemCond b))
28         (SemComm cmThen) s1 s2 → SemCond bPost s2
29     rThen = λ h → hypThen s1 s2 ((proj_2 (respAnd bPre b s1)) (q1 ,
30     proj_1 t1))
31     (proj_2 ((proj_2 (RelOpState.deltaRestPre (SemCond b) (SemComm
32     cmThen) s1 s2)) h))
33     rElse : RelOpState.comp (RelOpState.delta (NotP (SemCond b)))
34         (SemComm cmElse) s1 s2 → SemCond bPost s2
35     rElse = λ h →
36     let t10 : (NotP (SemCond b) s1) × (SemComm cmElse s1 s2
37     )
38     t10 = proj_2 (RelOpState.deltaRestPre
39     (NotP (SemCond b)) (SemComm cmElse) s1
40     s2) h
41     in hypElse s1 s2 (proj_2 (respAnd bPre (neg b) s1)
42     (q1 , (proj_2 (respNeg b s1) (proj_1 t10))))
43 (proj_2 t10)
44 in when rThen rElse q2
45 Soundness (WhileRule {cm'} {bInv} {b} pr) s1 s2 q1 q2
46 = proj_2 (respAnd bInv (neg b) s2)
47 (lem1 (proj_1 q2) s2 (proj_1 t15) , proj_2 (respNeg b s2) (proj
48 _2 t15))
49 where
50 hyp : Satisfies (bInv ∧ b) cm' bInv
51 hyp = Soundness pr
52 Rel1 : ℕ → Rel State (Level.zero)
53 Rel1 = λ m →
54     RelOpState.repeat
55     m
56     (RelOpState.comp (RelOpState.delta (SemCond b))
57     (SemComm cm'))
58 t15 : (Rel1 (proj_1 q2) s1 s2) × (NotP (SemCond b) s2)
59 t15 = proj_2 (RelOpState.deltaRestPost
60 (NotP (SemCond b)) (Rel1 (proj_1 q2)) s1 s2) (proj_2 q2)
61 lem1 : (m : ℕ) → (ss2 : State) → Rel1 m s1 ss2 → SemCond bInv
62 ss2
63 lem1 zero ss2 h = substId1 State (proj_2 h) (SemCond bInv) q1
64 lem1 (suc n) ss2 h
65 = let hyp2 : (z : State) → Rel1 (proj_1 q2) s1 z →
66     SemCond bInv z
67     hyp2 = lem1 n

```

```

60 |         t22 : (SemCond b (proj_1 h)) × (SemComm cm' (proj_1 h) ss2)
61 |         t22 = proj_2 (RelOpState.deltaRestPre (SemCond b) (SemComm
        cm') (proj_1 h) ss2)
62 |         (proj_2 (proj_2 h))
63 |         t23 : SemCond (bInv ∧ b) (proj_1 h)
64 |         t23 = proj_2 (respAnd bInv b (proj_1 h))
65 |         (hyp2 (proj_1 h) (proj_1 (proj_2 h)) , proj_1 t22)
66 |     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

```

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

```

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

```

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 `proof1` を `PrimSoundness` に入力として渡すことで満たすことができる。ソースコード 4.13 で、Agda 上の Hoare Logic を用いた `while program` を検証することができた。ここで書かれた健全性の検証の全体は付録のソースコード A-2 に載せる。

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 [17] 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

```
1 plus : {l : Level} {t : Set l} → (x y : ℕ) → (next : ℕ → t) → t
2 plus x zero next = next x
3 plus x (suc y) next = plus (suc x) y next
4
5 -- plus 10 20
6 --   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

```

1 data whileTestState : Set where
2   s1 : whileTestState
3   s2 : whileTestState
4   sf : whileTestState
5
6
7 whileTestStateP : whileTestState → EnvC → Set
8 whileTestStateP s1 env = (vari env ≡ 0) ∧ (varn env ≡ c10 env)
9 whileTestStateP s2 env = (varn env + vari env ≡ c10 env)
10 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性を返す 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

```

1 whileTestPwP : {l : Level} {t : Set l} → (c10 : ℕ) →
2   ((env : EnvC) → (mdg : (vari env ≡ 0) ∧ (varn env ≡ c10
3   env)) → t) → t
4 whileTestPwP c10 next = next env record { pi1 = refl ; pi2 = refl } where
5   env : EnvC
6   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 Meta DataGear を含んだ形で記述される。

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 program 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

```
1 whileTestP : {l : Level} {t : Set l} → (c10 : ℕ) → (Code : Envc → t)
  → t
2 whileTestP c10 next = next (record {c10 = c10 ; varn = c10 ; vari = 0 })
3
4 whileLoopP' : {l : Level} {t : Set l} → Envc → (next : Envc → t) → (
  exit : Envc → t) → t
5 whileLoopP' record { c10 = c10 ; varn = zero ; vari = vari } _ exit =
  exit record { c10 = c10 ; varn = zero ; vari = vari }
6 whileLoopP' record { c10 = c10 ; varn = suc varn1 ; vari = vari } next _
  = next (record {c10 = c10 ; varn = varn1 ; vari = suc vari })
7
8 {-# TERMINATING #-}
9 loopP : {l : Level} {t : Set l} → Envc → (exit : Envc → t) → t
10 loopP env exit = whileLoopP' env (λ env → loopP env exit ) exit
11
12 whileTestPCall : (c10 : ℕ) → Envc
13 whileTestPCall c10 = whileTestP {} {} c10 (λ env → loopP env (λ env
  → env))
14
15 -- whileTestPCall 10
16 -- 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'

```

1 loopP' : {l : Level} {t : Set l} → EnvC → (exit : EnvC → t) → t
2 loopP' record { c10 = c10 ; varn = zero ; vari = vari } exit =
3   exit (record { c10 = c10 ; varn = zero ; vari = vari })
4 loopP' record { c10 = c10 ; varn = (suc varn_1) ; vari = vari } exit =
5   whileLoopP' (record { c10 = c10 ; varn = (suc varn_1) ; vari = vari })
6   (λ env → loopP' (record { c10 = c10 ; varn = varn_1 ; vari = vari })
7     exit ) exit
7
8 whileTestPCall' : (c10 : ℕ) → EnvC
9 whileTestPCall' c10 = whileTestP' {_} {_} c10 (λ env → loopP' env (λ env
10   → env))
11 -- whileTestP' 10
12 -- 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。

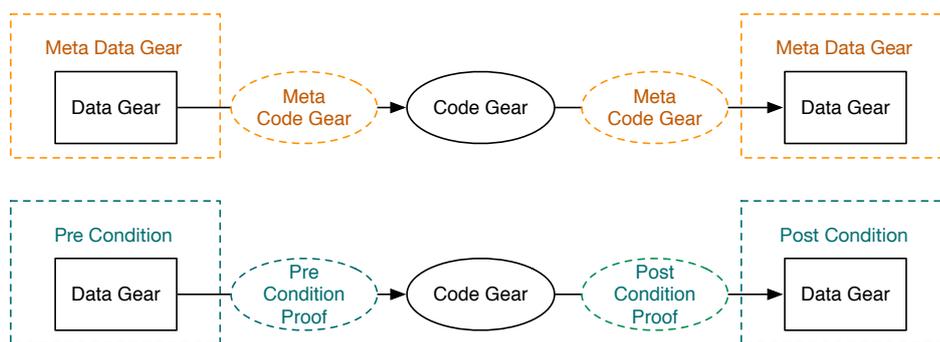


図 6.1: CodeGear、DataGear での Hoare Logic

ソースコード 6.3: CbC 上での Hoare Logic

```

1 -- Nomal CodeGear
2 whileLoop' : {l : Level} {t : Set l} → (n : ℕ) → (env : EnvC)
3   → (n ≡ varn env)
4   → (next : EnvC → t)
5   → (exit : EnvC → t) → t
6 whileLoop' zero env refl _ exit = exit env
7 whileLoop' (suc n) env refl next _ = next (record env {varn = pred (varn
8   env) ; vari = suc (vari env) })
9
10 -- Hoare Logic base CodeGear
11 whileLoopPwP' : {l : Level} {t : Set l} → (n : ℕ) → (env : EnvC )
12   → (n ≡ varn env) → (pre : varn env + vari env ≡ c10 env)
13   → (next : (env : EnvC) → (pred n ≡ varn env) → (post : varn env +
14     vari env ≡ c10 env) → t)
15   → (exit : (env : EnvC) → (fin : vari env ≡ c10 env) → t) → t
16 whileLoopPwP' zero env refl refl next exit = exit env refl
17 whileLoopPwP' (suc n) env refl refl next exit = next (record env {varn =
18   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ける事後条件となる。

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、事後条件を Meta DataGear とした Meta CodeGear を記述する必要がある。

6.3 CbC 上での Hoare Logic を用いた仕様記述

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

```

1 data whileTestState : Set where
2   s1 : whileTestState
3   s2 : whileTestState
4   sf : whileTestState
5
6 whileTestStateP : whileTestState → EnvC → Set
7 whileTestStateP s1 env = (vari env ≡ 0) ∧ (varn env ≡ c10 env)
8 whileTestStateP s2 env = (varn env + vari env ≡ c10 env)
9 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、sf 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 $(\text{vari env} \equiv 0) \wedge (\text{varn env} \equiv \text{c10 env})$ が成り立つ。

ソースコード 6.5: CbC 上の Hoare Logic での 代入

```

1 whileTestPwP : {l : Level} {t : Set l} → (c10 : ℕ) → ((env : EnvC) →
   whileTestStateP s1 env → t) → t
2 whileTestPwP c10 next = next env record { pi1 = refl ; pi2 = refl } where
3   env : EnvC
4   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、終わるの判断を行う 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 $(\text{varn env} + \text{vari env} \equiv \text{c10 env})$ の状態が成り立つ。varn が zero になると最後の loopPwP' に sf である $(\text{vari env} \equiv \text{c10 env})$ 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 $(\text{vari env} \equiv \text{c10 env})$ 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

```

1 whileLoopPwP' : {l : Level} {t : Set l} → (n : ℕ) → (env : EnvC)

```

```

2 |   → (n ≡ varn env) → whileTestStateP s2 env
3 |   → (next : (env : EnvC) → (pred n ≡ varn env) → whileTestStateP s2
   env → t)
4 |   → (exit : (env : EnvC) → whileTestStateP sf env → t) → t
5 | whileLoopPwP' zero env refl refl _ exit = exit env refl
6 | whileLoopPwP' (suc n) env refl refl next _ =
7 |   next (record env {varn = pred (varn env) ; vari = suc (vari env) })
   refl (+-suc n (vari env))
8 |
9 |
10 | loopPwP' : {l : Level} {t : Set l} → (n : ℕ) → (env : EnvC)
11 |   → (n ≡ varn env) → whileTestStateP s2 env
12 |   → (exit : (env : EnvC) → whileTestStateP sf env → t) → t
13 | loopPwP' zero env refl refl exit = exit env refl
14 | loopPwP' (suc n) env refl refl exit
15 | = whileLoopPwP' (suc n) env refl refl (λ env x y → loopPwP' n env x
   y exit) exit

```

これらの Meta CodeGear を使い仕様を記述する。

ソースコード 6.7: CbC 上の Hoare Logic

```

1 | whileCallwP : (c : ℕ) → Set
2 | whileCallwP c = whileTestPwP {_} {_} c (λ env s → loopPwP' (varn env)
   env refl (conv env s) (λ env s → vari env ≡ c10 env) )
3 | where
4 |   conv : (env : EnvC) → (vari env ≡ 0) ∧ (varn env ≡ c10 env) →
   varn env + vari env ≡ c10 env
5 |   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。

`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。

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 \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する必要がある。ソースコード 6.8 の `{!!}` は Agda の ? と同義で中に入る値が不明な状態である。

ソースコード 6.8: 接続途中の CbC 上での CodeGear の条件接続

```

1 | whileCallwP : (c : ℕ) → whileTestPCallwP' c
2 | whileCallwP c = whileTestPwP {_} {_} c
3 |   (λ env s → loopPwP' (c10 env) env (sym (pi2 s)) (conv env s) {!!})
4 | where

```

```

5   conv : (env : EnvC ) → (vari env ≡ 0) ∧ (varn env ≡ c10 env) →
6   varn env + vari env ≡ c10 env
   conv e record { pi1 = refl ; pi2 = refl } = +zero

```

whileTestPwP は代入のため単純に Agda が計算できる。しかし、loopPwP' は実際の値が入るまで計算をすすめることができず、条件を接続することができなかった。

そのためソースコード 6.9 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

```

1 loopHelper : (n : ℕ) → (env : EnvC ) → (eq : varn env ≡ n) → (seq :
   whileTestStateP s2 env)
2   → loopPwP' n env (sym eq) seq (λ env_1 x → (vari env_1 ≡ c10
   env_1))
3 loopHelper zero env eq refl rewrite eq = refl
4 loopHelper (suc n) env refl refl =
5   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。

loopHelper を使い loopPwP のループでの停止性を示すことができたため、ソースコード 6.8 のすべての条件が繋げられる。

ソースコード 6.10 は完成した CbC 上での Hoare Logic ベースの

ソースコード 6.10: 完成した CbC 上での CodeGear の条件接続

```

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

```

これにより、すべての条件を接続した仕様である ソースコード 6.10 を構築できた。

6.4 CbC 上での Hoare Logic を用いた検証の健全性

本章では CbC 上の Hoare Logic での健全性を含めた検証する。

CbC 上での Hoare Logic の健全性では、仕様を含んだ Meta CodeGear が正しく接続できることと、各 CodeGear が事前条件から事後条件へ遷移することを満たしている必要がある。

また、各 CodeGear が正しく停止することも重要である。

6.3 の ソースコード 6.10 では仕様が正しく接続でき、それぞれの CodeGear が停止することも分かっている。

ここでは CodeGear が正しく条件を遷移させることができることを示す必要がある。

健全性を含めた検証のために `implies` という data を導入する。

ソースコード 6.11 は `implies` の定義である。 `implies` の型は A と B の条件を受け取り、 $A \text{ implies } B$ の形で型として記述する。このとき `proof` として $A \rightarrow B$ が存在すれば $A \text{ implies } B$ が証明でき、 A から B へと正しく遷移できることが示せる。

ソースコード 6.11: `implies`

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

この `implies` を用いて while program で使用している CodeGear が事前条件から正しく事後条件に遷移することを証明する。

例として、代入を行う CodeGear である `whileTestP` の検証を行う。ソースコード 6.12 は `whileTestP` の検証の CodeGear である。 `whileTestP` を実行したときに、常に真の命題 \top と代入を終えたときの事後条件にあたる $(\text{vari env} \equiv 0) \wedge (\text{varn env} \equiv \text{c10 env})$ を `implies` に入れた型を記述している。

ソースコード 6.12: 代入を行う `whileTestP` の健全性

```
1 whileTestPSemSound : (c : ℕ) (output : EnvC)
2   → output ≡ whileTestP c (λ e → e)
3   → ⊤ implies ((vari output ≡ 0) ∧ (varn output ≡ c))
4   whileTestPSemSound c output refl = proof (λ _ → record { pi1 = refl ;
   pi2 = refl })
```

証明では $(\top \rightarrow (\text{vari env} \equiv 0) \wedge (\text{varn env} \equiv \text{c10 env}))$ を示す必要があり、ここでは `proof` の引数に入っている $(\lambda _ \rightarrow \text{record } \{ \text{pi1} = \text{refl} ; \text{pi2} = \text{refl} \})$ が対応する。

同様に他の CodeGear に対しても健全性の証明を行う。

ソースコード 6.13 の `whileConvPSemSound` は制約を緩める `conversion` の健全性の証明である。

ソースコード 6.13: 条件を変更する `Conversion` の健全性

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

ソースコード 6.14 の `whileLoopPSemSound` はループを行う CodeGear の健全性の証明である。

ソースコード 6.14: while loop をする `loopPP` の健全性

```
1 loopPPSem : (input output : EnvC) → output ≡ loopPP (varn input)
   input refl
```

```

2 | → (whileTestStateP s2 input ) → (whileTestStateP s2 input ) implies (
   | whileTestStateP sf output)
3 | loopPPSem input output refl s2p = loopPPSemInduct (varn input) input
   | refl refl s2p
4 | where
5 |   lem : (n : ℕ) → (env : Eenvc) → n + suc (vari env) ≡ suc (n + vari
   | env)
6 |   lem n env = +-suc (n) (vari env)
7 |   loopPPSemInduct : (n : ℕ) → (current : Eenvc) → (eq : n ≡ varn
   | current) → (loopeq : output ≡ loopPP n current eq)
8 |     → (whileTestStateP s2 current ) → (whileTestStateP s2 current )
   | implies (whileTestStateP sf output)
9 |   loopPPSemInduct zero current refl loopeq refl rewrite loopeq = proof
   | (λ x → refl)
10 |  loopPPSemInduct (suc n) current refl loopeq refl rewrite (sym (lem n
   | current)) =
11 |     whileLoopPSem current refl
12 |     (λ output x → loopPPSemInduct n (record { c10 = n + suc (
   | vari current) ; varn = n ; vari = suc (vari current) }) refl loopeq
   | refl)
13 |     (λ output x → loopPPSemInduct n (record { c10 = n + suc (
   | vari current) ; varn = n ; vari = suc (vari current) }) refl loopeq
   | refl)
14 |
15 |
16 | whileLoopPSemSound : {l : Level} → (input output : Eenvc )
17 |   → (varn input + vari input ≡ c10 input)
18 |   → output ≡ loopPP (varn input) input refl
19 |   → (varn input + vari input ≡ c10 input) implies (vari output ≡ c10
   | output)
20 | whileLoopPSemSound {l} input output pre eq = loopPPSem input output eq
   | pre

```

ソースコード 6.15 は loopPP の証明のための補助定理で、ループの判定を行う whileLoopPSem とループを行う loopPPSem に分かれている。

whileLoopPSem は行き先としてループの継続である next とループの終了を行う exit の 2 つの Meta CodeGear を持つ。whileLoopPSem では現在の Eenvc の値を確認し、それぞれの状態が正しく状態を遷移しているかを確認する。

loopPPSem は実際に while loop する Meta CodeGear で、ループの終了である exit の Meta CodeGear しか持たない。これは loopPPSem では最終的にループが終了したときの遷移が示せば良いためである。

loopPPSem は loopPPSemInduct という別の MetaCodeGear を使ってループをしており、このループが varn の値だけループを繰り返すことを記述しており、停止性を明示している。

ソースコード 6.15: loopPP を証明するための補助定理 loopPPSem と whileLoopPSem

```

1 | whileLoopPSem : {l : Level} {t : Set l} → (input : Eenvc ) → (vari
   | input) + (varn input) ≡ (c10 input)

```

```

2 | → (next : (output : Eenvc) → ((vari input) + (varn input) ≡ (c10 input)
   |   ) implies ((vari output) + (varn output) ≡ (c10 output)) → t)
3 | → (exit : (output : Eenvc) → ((vari input) + (varn input) ≡ (c10 input)
   |   ) implies ((vari output ≡ c10 output)) → t) → t
4 | whileLoopPSem env s next exit with varn env | s
5 | ... | zero | _ = exit env (proof (λ z → z))
6 | ... | (suc varn) | refl = next ( record env { varn = varn ; vari = suc (
   |   vari env) } ) (proof λ x → +-suc varn (vari env) )
7 |
8 |
9 | loopPPSem : (input output : Eenvc) → output ≡ loopPP (varn input)
   |   input refl
10 | → (vari input) + (varn input) ≡ (c10 input) → ((vari input) + (varn
   |   input) ≡ (c10 input)) implies ((vari output ≡ c10 output))
11 | loopPPSem input output refl s2p = loopPPSemInduct (varn input) input
   |   refl refl s2p
12 | where
13 |   lem : (n : ℕ) → (env : Eenvc) → n + suc (vari env) ≡ suc (n + vari
   |   env)
14 |   lem n env = +-suc (n) (vari env)
15 |   loopPPSemInduct : (n : ℕ) → (current : Eenvc) → (eq : n ≡ varn
   |   current) → (loopeq : output ≡ loopPP n current eq)
16 |   → ((vari current) + (varn current) ≡ (c10 current)) → ((vari
   |   current) + (varn current) ≡ (c10 current)) implies ((vari output ≡
   |   c10 output))
17 |   loopPPSemInduct zero current refl loopeq refl rewrite loopeq = proof
   |   (λ x → refl)
18 |   loopPPSemInduct (suc n) current refl loopeq refl rewrite (sym (lem n
   |   current)) =
19 |     whileLoopPSem current refl
20 |     (λ output x → loopPPSemInduct n (record { c10 = n + suc (
   |   vari current) ; varn = n ; vari = suc (vari current) }) refl loopeq
   |   refl)
21 |     (λ output x → loopPPSemInduct n (record { c10 = n + suc (
   |   vari current) ; varn = n ; vari = suc (vari current) }) refl loopeq
   |   refl)

```

これで while program で使用されている CodeGear すべてが正しく遷移することを証明できた。

これらの CodeGear は仕様記述ですべての CodeGear と条件が接続できており、`implies` で証明した対応する遷移が正しいことから、CbC の Hoare Logic をベースとして書かれた while program は健全であると言える。

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を行った。また、`implies` を用いて状態遷移を記述することで CbC 上での Hoare Logic の健全性を示すことができた。

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、CbC 上での Pre Condition、Post Condition の記述方法が明確になった。

また、検証時に任意の回数ループする Meta CodeGear が存在するとき、任意の自然数回ループしても正しく次の状態に遷移できることを証明することで、証明が記述できることが判明した。

さらに、本来の Hoare Logic では決められた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。

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 Red-Black Tree などのデータ構造が挙げられる。

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。

CodeGear の健全性に関しては、コマンドが限定されていた Hoare Logic とは異なり、`implies` を用いて CodeGear に対してそれぞれの遷移系を記述する必要がある。これらの遷移系は Meta CodeGear として記述できているため、仕様記述から変換できる形になることが期待できる。

現在の CbC では、検証の記述を Agda で記述しているため、実際の実行に向く記述ではない。そのため、検証を行った Agda 上の CbC 記述から Agda を用いて、より高速に実行できる CbC プログラムの生成を行う必要がある。

また、CbC で開発、検証したいと考えている GearsOS には並列構文が存在しており、どのように並列動作に対しての検証も課題として挙げられる。

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. Llm 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] Jean-Yves Girard, Paul Taylor, and Yves Lafont. Proofs and Types. Cambridge University Press, USA, 1989.
- [13] C. A. R. Hoare. An axiomatic basis for computer programming. Commun. ACM, Vol. 12, No. 10, p. 576–580, October 1969.
- [14] Agda1. <https://sourceforge.net/projects/agda>. Accessed: 2020/2/9(Sun).
- [15] Hoare logic in agda2. <https://github.com/IKEGAMIDaisuke/HoareLogic>. Accessed: 2020/2/9(Sun).
- [16] Andreas Abel. foetus – termination checker for simple functional programs, 1998.
- [17] 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.
- [18] 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.
- [19] Eugenio Moggi. Notions of computation and monads. Inf. Comput., Vol. 93, No. 1, pp. 55–92, July 1991.
- [20] 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.
- [21] 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.
- [22] The agda wiki. <http://wiki.portal.chalmers.se/agda/pmwiki.php>. Accessed: 2018/12/17(Mon).
- [23] Welcome to agda’ s documentation! — agda latest documentation. <http://agda.readthedocs.io/en/latest/>. Accessed: 2018/12/17(Mon).
- [24] Aaron Stump. Verified Functional Programming in Agda. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2016.
- [25] Example - hoare logic. <http://ocvs.cfv.jp/Agda/readmehoare.html>. Accessed: 2019/1/16(Wed).

- [26] Welcome! — the coq proof assistant. <https://coq.inria.fr>. Accessed: 2019/1/16(Wed).
- [27] Hoare logic - 並列信頼研 mercurial repository. <http://www.cr.ie.u-ryukyu.ac.jp/hg/Members/ryokka/HoareLogic/>. Accessed: 2020/2/9(Sun).
- [28] 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.
- [29] 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.
- [30] 村井哲也, 深海悟. 様相論理 1. 日本フエジィ学会誌, Vol. 7, No. 1, pp. 3–18, 1995.

付録A ソースコード一覧

本論文中で取り上げた Agda の全体のソースコードを示す。確認した Agda のバージョンは 2.6.0.1 で standard library は 1.2 である。

A-1 Agda 上の Hoare Logic のコマンドで記述された while program (whileTestPrim.agda)

Hoare Logic を実際に使って記述した while program とコマンドを解釈して実行するための interpret を ソースコード A.1 に載せる。Agda 上の Hoare Logic の定義は、現在の Agda に直した Hoare Logic の定義 [15] の Hoare.agda に記述されているものと同様である。

ソースコード A.1: Agda 上での Hoare Logic を用いた while program と検証

```
1 module whileTestPrim where
2
3 open import Function
4 open import Data.Nat
5 open import Data.Bool hiding ( _≐_ )
6 open import Level renaming ( suc to succ ; zero to Zero )
7 open import Relation.Nullary using ( ¬_ ; Dec ; yes ; no )
8 open import Relation.Binary.PropositionalEquality
9
10 open import utilities hiding ( _^_ )
11
12 record Env : Set where
13   field
14     varn : ℕ
15     vari : ℕ
16 open Env
17
18 PrimComm : Set
19 PrimComm = Env → Env
20
21 Cond : Set
22 Cond = (Env → Bool)
23
24 Axiom : Cond → PrimComm → Cond → Set
```

```

25 | Axiom pre comm post =  $\forall$  (env : Env)  $\rightarrow$  (pre env)  $\Rightarrow$  ( post (comm env))
    |  $\equiv$  true
26 |
27 | Tautology : Cond  $\rightarrow$  Cond  $\rightarrow$  Set
28 | Tautology pre post =  $\forall$  (env : Env)  $\rightarrow$  (pre env)  $\Rightarrow$  (post env)  $\equiv$  true
29 |
30 | _and_ : Cond  $\rightarrow$  Cond  $\rightarrow$  Cond
31 | x and y =  $\lambda$  env  $\rightarrow$  x env  $\wedge$  y env
32 |
33 | neg : Cond  $\rightarrow$  Cond
34 | neg x =  $\lambda$  env  $\rightarrow$  not ( x env )
35 |
36 | open import Hoare PrimComm Cond Axiom Tautology _and_ neg
37 |
38 | -----
39 |
40 | program :  $\mathbb{N} \rightarrow$  Comm
41 | program c10 =
42 |   Seq ( PComm ( $\lambda$  env  $\rightarrow$  record env {varn = c10}))
43 |   $ Seq ( PComm ( $\lambda$  env  $\rightarrow$  record env {vari = 0}))
44 |   $ While ( $\lambda$  env  $\rightarrow$  lt zero (varn env) )
45 |     (Seq (PComm ( $\lambda$  env  $\rightarrow$  record env {vari = ((vari env) + 1)} ))
46 |       $ PComm ( $\lambda$  env  $\rightarrow$  record env {varn = ((varn env) - 1)} ))
47 |
48 | simple :  $\mathbb{N} \rightarrow$  Comm
49 | simple c10 =
50 |   Seq ( PComm ( $\lambda$  env  $\rightarrow$  record env {varn = c10}))
51 |   $ PComm ( $\lambda$  env  $\rightarrow$  record env {vari = 0})
52 |
53 | {-# TERMINATING #-}
54 | interpret : Env  $\rightarrow$  Comm  $\rightarrow$  Env
55 | interpret env Skip = env
56 | interpret env Abort = env
57 | interpret env (PComm x) = x env
58 | interpret env (Seq comm comm1) = interpret (interpret env comm) comm1
59 | interpret env (If x then else) with x env
60 | ... | true = interpret env then
61 | ... | false = interpret env else
62 | interpret env (While x comm) with x env
63 | ... | true = interpret (interpret env comm) (While x comm)
64 | ... | false = env
65 |
66 | test1 : Env
67 | test1 = interpret ( record { vari = 0 ; varn = 0 } ) (program 10)
68 |
69 |
70 | tests : Env
71 | tests = interpret ( record { vari = 0 ; varn = 0 } ) (simple 10)

```

A-2 Hoare Logic での while program の検証 (whileTestPrimProof.agda)

実際に while program を Hoare Logic で検証したコードをソースコード A.2 に載せる。Hoare Logic のコマンドの仕様や、Satisfies、Soundness などの定義は、Hoare Logic の定義 [15] の Hoare.agda や HoareSoundness.agda、RelOp.agda と同様である。

ソースコード A.2: Agda 上での Hoare Logic を用いた while program の検証

```

1 module whileTestPrimProof where
2
3 open import Function
4 open import Data.Nat
5 open import Data.Bool hiding ( _≡_ )
6 open import Level renaming ( suc to succ ; zero to Zero )
7 open import Relation.Nullary using ( ¬_ ; Dec ; yes ; no )
8 open import Relation.Binary.PropositionalEquality
9
10 open import utilities hiding ( _^_ )
11 open import whileTestPrim
12
13 open import Hoare PrimComm Cond Axiom Tautology _and_ neg
14
15 open Env
16
17 initCond : Cond
18 initCond env = true
19
20 stmt1Cond : {c10 : ℕ} → Cond
21 stmt1Cond {c10} env = Equal (varn env) c10
22
23 init-case : {c10 : ℕ} → (env : Env) → (( λ e → true ⇒ stmt1Cond {
24   c10 } e ) (record { varn = c10 ; vari = vari env } ) ) ≡ true
25
26 init-case {c10} _ = impl⇒ ( λ cond → ≡→Equal refl )
27
28 init-type : {c10 : ℕ} → Axiom (λ env → true) (λ env → record { varn =
29   c10 ; vari = vari env } ) (stmt1Cond {c10})
30
31 init-type {c10} env = init-case env
32
33 stmt2Cond : {c10 : ℕ} → Cond
34 stmt2Cond {c10} env = (Equal (varn env) c10) ∧ (Equal (vari env) 0)
35
36 lemma1 : {c10 : ℕ} → Axiom (stmt1Cond {c10}) (λ env → record { varn =
37   varn env ; vari = 0 } ) (stmt2Cond {c10})
38
39 lemma1 {c10} env = impl⇒ ( λ cond → let open ≡-Reasoning in
40   begin
41     (Equal (varn env) c10 ) ∧ true
42   ≡⟨ ∧true ⟩
43     Equal (varn env) c10
44   ≡⟨ cond ⟩
45     true

```

```

40 |   )
41 |
42 | -- simple : ℕ → Comm
43 | -- simple c10 =
44 | --   Seq ( PComm (λ env → record env {varn = c10}))
45 | --   $ PComm (λ env → record env {vari = 0})
46 |
47 | proofs : (c10 : ℕ) → HTProof initCond (simple c10) (stmt2Cond {c10})
48 | proofs c10 =
49 |   SeqRule {initCond} ( PrimRule (init-case {c10} ))
50 |   $ PrimRule {stmt1Cond} { _ } {stmt2Cond} (lemma1 {c10})
51 |
52 | open import Data.Empty
53 |
54 | open import Data.Nat.Properties
55 |
56 | whileInv : {c10 : ℕ} → Cond
57 | whileInv {c10} env = Equal ((varn env) + (vari env)) c10
58 |
59 | whileInv' : {c10 : ℕ} → Cond
60 | whileInv' {c10} env = Equal ((varn env) + (vari env)) (suc c10) ∧ lt zero
   |   (varn env)
61 |
62 | termCond : {c10 : ℕ} → Cond
63 | termCond {c10} env = Equal (vari env) c10
64 |
65 |
66 | -- program : ℕ → Comm
67 | -- program c10 =
68 | --   Seq ( PComm (λ env → record env {varn = c10}))
69 | --   $ Seq ( PComm (λ env → record env {vari = 0}))
70 | --   $ While (λ env → lt zero (varn env) )
71 | --   (Seq (PComm (λ env → record env {vari = ((vari env) + 1)} ))
72 | --   $ PComm (λ env → record env {varn = ((varn env) - 1)} ))
73 |
74 |
75 | proof1 : (c10 : ℕ) → HTProof initCond (program c10 ) (termCond {c10})
76 | proof1 c10 =
77 |   SeqRule {λ e → true} ( PrimRule (init-case {c10} ))
78 |   $ SeqRule {λ e → Equal (varn e) c10} ( PrimRule lemma1 )
79 |   $ WeakeningRule {λ e → (Equal (varn e) c10) ∧ (Equal (vari e) 0)}
   |   lemma2 (
80 |     WhileRule { _ } {λ e → Equal ((varn e) + (vari e)) c10}
81 |     $ SeqRule (PrimRule {λ e → whileInv e ∧ lt zero (varn e) }
   |     lemma3 )
82 |     $ PrimRule {whileInv'} { _ } {whileInv} lemma4 )
   |   lemma5
83 | where
84 |   lemma21 : {env : Env } → {c10 : ℕ} → stmt2Cond env ≡ true → varn
   |   env ≡ c10
85 |   lemma21 eq = Equal→≡ (∧-pi1 eq)
86 |   lemma22 : {env : Env } → {c10 : ℕ} → stmt2Cond {c10} env ≡ true
   |   → vari env ≡ 0

```

```

87 lemma22 eq = Equal → ≡ (∧-pi2 eq)
88 lemma23 : {env : Env } → {c10 : ℕ} → stmt2Cond env ≡ true →
varn env + vari env ≡ c10
89 lemma23 {env} {c10} eq = let open ≡-Reasoning in
90 begin
91   varn env + vari env
92   ≡⟨ cong ( \ x → x + vari env ) (lemma21 eq ) ⟩
93     c10 + vari env
94   ≡⟨ cong ( \ x → c10 + x ) (lemma22 {env} {c10} eq ) ⟩
95     c10 + 0
96   ≡⟨ +-sym {c10} {0} ⟩
97     0 + c10
98   ≡⟨ ⟩
99     c10
100   ■
101 lemma2 : {c10 : ℕ} → Tautology stmt2Cond whileInv
102 lemma2 {c10} env = bool-case (stmt2Cond env) (
103   λ eq → let open ≡-Reasoning in
104     begin
105       (stmt2Cond env) ⇒ (whileInv env)
106     ≡⟨ ⟩
107       (stmt2Cond env) ⇒ ( Equal (varn env + vari env) c10 )
108     ≡⟨ cong ( \ x → (stmt2Cond {c10} env) ⇒ ( Equal x c10 ) )
( lemma23 {env} eq ) ⟩
109       (stmt2Cond env) ⇒ (Equal c10 c10)
110     ≡⟨ cong ( \ x → (stmt2Cond {c10} env) ⇒ x ) (≡→Equal refl )
111     ⟩
112       (stmt2Cond {c10} env) ⇒ true
113     ≡⟨ ⇒t ⟩
114       true
115     ■
116   ) (
117     λ ne → let open ≡-Reasoning in
118       begin
119         (stmt2Cond env) ⇒ (whileInv env)
120       ≡⟨ cong ( \ x → x ⇒ (whileInv env) ) ne ⟩
121         false ⇒ (whileInv {c10} env)
122       ≡⟨ f⇒ {whileInv {c10} env} ⟩
123         true
124       ■
125     )
126 lemma3 : Axiom (λ e → whileInv e ∧ lt zero (varn e)) (λ env →
record { varn = varn env ; vari = vari env + 1 }) whileInv'
127 lemma3 env = impl⇒ ( λ cond → let open ≡-Reasoning in
128   begin
129     whileInv' (record { varn = varn env ; vari = vari env + 1 })
130   ≡⟨ ⟩
131     Equal (varn env + (vari env + 1)) (suc c10) ∧ (lt 0 (varn
env) )
132   ≡⟨ cong ( λ z → Equal (varn env + (vari env + 1)) (suc c10) ∧
z ) (∧-pi2 cond ) ⟩
133     Equal (varn env + (vari env + 1)) (suc c10) ∧ true

```

```

133     ≡⟨  $\wedge$ true ⟩
134     Equal (varn env + (vari env + 1)) (suc c10)
135     ≡⟨ cong (  $\lambda$  x → Equal x (suc c10) ) (sym (+-assoc (varn env) (
136     vari env) 1)) ⟩
137     Equal ((varn env + vari env) + 1) (suc c10)
138     ≡⟨ cong (  $\lambda$  x → Equal x (suc c10) ) +1≡suc ⟩
139     Equal (suc (varn env + vari env)) (suc c10)
140     ≡⟨ sym Equal+1 ⟩
141     Equal ((varn env + vari env) ) c10
142     ≡⟨  $\wedge$ -pi1 cond ⟩
143     true
144     ■ )
145 lemma41 : (env : Env) → {c10 : ℕ} → (varn env + vari env) ≡ (suc
146 c10) → lt 0 (varn env) ≡ true → Equal ((varn env - 1) + vari env)
147 c10 ≡ true
148 lemma41 env {c10} c1 c2 = let open ≡-Reasoning in
149 begin
150   Equal ((varn env - 1) + vari env) c10
151   ≡⟨ cong (  $\lambda$  z → Equal ((z - 1) + vari env) c10 ) (sym (suc-
152   predℕ=n c2) ) ⟩
153   Equal ((suc (predℕ {varn env} c2) - 1) + vari env) c10
154   ≡⟨ ⟩
155   Equal ((predℕ {varn env} c2) + vari env) c10
156   ≡⟨ Equal+1 ⟩
157   Equal ((suc (predℕ {varn env} c2)) + vari env) (suc c10)
158   ≡⟨ cong (  $\lambda$  z → Equal (z + vari env) (suc c10) ) (suc-pred
159   ℕ=n c2) ) ⟩
160   Equal (varn env + vari env) (suc c10)
161   ≡⟨ cong (  $\lambda$  z → (Equal z (suc c10) )) c1 ⟩
162   Equal (suc c10) (suc c10)
163   ≡⟨  $\equiv$ →Equal refl ⟩
164   true
165   ■
166 lemma4 : {c10 : ℕ} → Axiom whileInv' ( $\lambda$  env → record { varn =
167 varn env - 1 ; vari = vari env }) whileInv
168 lemma4 {c10} env = impl⇒ (  $\lambda$  cond → let open ≡-Reasoning in
169 begin
170   whileInv (record { varn = varn env - 1 ; vari = vari env })
171   ≡⟨ ⟩
172   Equal ((varn env - 1) + vari env) c10
173   ≡⟨ lemma41 env (Equal→≡ ( $\wedge$ -pi1 cond)) ( $\wedge$ -pi2 cond) ⟩
174   true
175   ■
176 )
177 lemma51 : (z : Env) → neg ( $\lambda$  z → lt zero (varn z)) z ≡ true →
178 varn z ≡ zero
179 lemma51 z cond with varn z
180 lemma51 z refl | zero = refl
181 lemma51 z () | suc x
182 lemma5 : {c10 : ℕ} → Tautology (( $\lambda$  e → Equal (varn e + vari e)
183 c10) and (neg ( $\lambda$  z → lt zero (varn z)))) termCond
184 lemma5 {c10} env = impl⇒ (  $\lambda$  cond → let open ≡-Reasoning in

```

```

177 |         begin
178 |           termCond env
179 |           ≡⟨
180 |             Equal (vari env) c10
181 |           ≡⟨
182 |             Equal (zero + vari env) c10
183 |           ≡⟨ cong ( λ z → Equal (z + vari env) c10 ) (sym ( lemma51
env ( ^-pi2 cond ) )) ⟩
184 |             Equal (varn env + vari env) c10
185 |           ≡⟨ ^-pi1 cond ⟩
186 |             true
187 |           ■
188 |         )
189 |
190 | --- necessary definitions for Hoare.agda ( Soundness )
191 |
192 | State : Set
193 | State = Env
194 |
195 | open import RelOp
196 | module RelOpState = RelOp State
197 |
198 | open import Data.Product
199 | open import Relation.Binary
200 |
201 | NotP : {S : Set} → Pred S → Pred S
202 | NotP X s = ¬ X s
203 |
204 | _^_ : Cond → Cond → Cond
205 | b1 ^ b2 = b1 and b2
206 |
207 | _\/_ : Cond → Cond → Cond
208 | b1 \/_ b2 = neg (neg b1 ^ neg b2)
209 |
210 | SemCond : Cond → State → Set
211 | SemCond c p = c p ≡ true
212 |
213 | tautValid : (b1 b2 : Cond) → Tautology b1 b2 →
214 |             (s : State) → SemCond b1 s → SemCond b2 s
215 | tautValid b1 b2 taut s cond with b1 s | b2 s | taut s
216 | tautValid b1 b2 taut s () | false | false | refl
217 | tautValid b1 b2 taut s _ | false | true | refl = refl
218 | tautValid b1 b2 taut s _ | true | false | ()
219 | tautValid b1 b2 taut s _ | true | true | refl = refl
220 |
221 | respNeg : (b : Cond) → (s : State) →
222 |           Iff (SemCond (neg b) s) (¬ SemCond b s)
223 | respNeg b s = ( left , right ) where
224 |   left : not (b s) ≡ true → (b s) ≡ true → ⊥
225 |   left ne with b s
226 |   left refl | false = λ ()
227 |   left () | true
228 |   right : ((b s) ≡ true → ⊥) → not (b s) ≡ true
229 |   right ne with b s

```

```

230 |     right ne | false = refl
231 |     right ne | true =  $\perp$ -elim ( ne refl )
232 |
233 | respAnd : (b1 b2 : Cond) → (s : State) →
234 |           Iff (SemCond (b1  $\wedge$  b2) s)
235 |             ((SemCond b1 s)  $\times$  (SemCond b2 s))
236 | respAnd b1 b2 s = ( left , right ) where
237 |   left : b1 s  $\wedge$  b2 s  $\equiv$  true → (b1 s  $\equiv$  true)  $\times$  (b2 s  $\equiv$  true)
238 |   left and with b1 s | b2 s
239 |   left () | false | false
240 |   left () | false | true
241 |   left () | true | false
242 |   left refl | true | true = ( refl , refl )
243 |   right : (b1 s  $\equiv$  true)  $\times$  (b2 s  $\equiv$  true) → b1 s  $\wedge$  b2 s  $\equiv$  true
244 |   right ( x1 , x2 ) with b1 s | b2 s
245 |   right ((), ()) | false | false
246 |   right ((), _) | false | true
247 |   right (_, ()) | true | false
248 |   right (refl , refl) | true | true = refl
249 |
250 | PrimSemComm :  $\forall$  {l} → PrimComm → Rel State l
251 | PrimSemComm prim s1 s2 = Id State (prim s1) s2
252 |
253 |
254 |
255 | axiomValid :  $\forall$  {l} → (bPre : Cond) → (pcm : PrimComm) → (bPost : Cond)
256 |           →
257 |             (ax : Axiom bPre pcm bPost) → (s1 s2 : State) →
258 |               SemCond bPre s1 → PrimSemComm {l} pcm s1 s2 → SemCond
259 |                 bPost s2
260 | axiomValid {l} bPre pcm bPost ax s1 .(pcm s1) semPre ref with bPre s1 |
261 |   bPost (pcm s1) | ax s1
262 | axiomValid {l} bPre pcm bPost ax s1 .(pcm s1) () ref | false | false |
263 |   refl
264 | axiomValid {l} bPre pcm bPost ax s1 .(pcm s1) semPre ref | false | true |
265 |   refl = refl
266 | axiomValid {l} bPre pcm bPost ax s1 .(pcm s1) semPre ref | true | false |
267 |   ()
268 | axiomValid {l} bPre pcm bPost ax s1 .(pcm s1) semPre ref | true | true |
269 |   refl = refl
270 |
271 | open import HoareSoundness
272 |   Cond
273 |   PrimComm
274 |   neg
275 |   _and_
276 |   Tautology
277 |   State
278 |   SemCond
279 |   tautValid
280 |   respNeg
281 |   respAnd
282 |   PrimSemComm
283 |   Axiom

```

```

277 |     axiomValid
278 |
279 | PrimSoundness : {bPre : Cond} → {cm : Comm} → {bPost : Cond} →
280 |     HTProof bPre cm bPost → Satisfies bPre cm bPost
281 | PrimSoundness {bPre} {cm} {bPost} ht = Soundness ht
282 |
283 |
284 | proofOfProgram : (c10 : ℕ) → (input output : Env )
285 |     → initCond input ≡ true
286 |     → (SemComm (program c10) input output)
287 |     → termCond {c10} output ≡ true
288 | proofOfProgram c10 input output ic sem = PrimSoundness (proof1 c10)
    input output ic sem

```

A-3 CbC の Hoare Logic をベースにした while program の仕様記述と検証 (whileTestGears.agda)

ソースコード A.3 に CbC 上での Hoare Logic を用いた while program とその検証を行ったコードの全文を載せる。

ソースコード A.3: Agda 上での Hoare Logic を用いた while program と検証

```

1 | module whileTestGears where
2 |
3 | open import Function
4 | open import Data.Nat
5 | open import Data.Bool hiding ( _≟_ ; _≤?_ ; _≤_ ; _<_ )
6 | open import Data.Product
7 | open import Level renaming ( succ to succ ; zero to Zero )
8 | open import Relation.Nullary using ( ¬_ ; Dec ; yes ; no )
9 | open import Relation.Binary.PropositionalEquality
10 | open import Agda.Builtin.Unit
11 |
12 | open import utilities
13 | open _^_
14 |
15 | -- original codeGear (with non terminating )
16 |
17 | record Env : Set (succ Zero) where
18 |   field
19 |     varn : ℕ
20 |     vari : ℕ
21 | open Env
22 |
23 | whileTest : {l : Level} {t : Set l} → (c10 : ℕ) → (Code : Env → t)
    → t
24 | whileTest c10 next = next (record {varn = c10 ; vari = 0 } )
25 |
26 | {-# TERMINATING #-}

```

```

27 | whileLoop : {l : Level} {t : Set l} → Env → (Code : Env → t) → t
28 | whileLoop env next with lt 0 (varn env)
29 | whileLoop env next | false = next env
30 | whileLoop env next | true =
31 |   whileLoop (record env {varn = (varn env) - 1 ; vari = (vari env) +
32 |     1}) next
33 | test1 : Env
34 | test1 = whileTest 10 (λ env → whileLoop env (λ env1 → env1 ))
35 |
36 | proof1 : whileTest 10 (λ env → whileLoop env (λ e → (vari e) ≡ 10 ))
37 | proof1 = refl
38 |
39 | -- codeGear with pre-condtion and post-condition
40 | --
41 | --
42 |   ↓
43 |   PostCondition
44 | whileTest' : {l : Level} {t : Set l} → {c10 : ℕ} → (Code : (env : Env
45 |   ) → ((vari env) ≡ 0) ∧ ((varn env) ≡ c10) → t) → t
46 | whileTest' [_] [_] {c10} next = next env proof2
47 | where
48 |   env : Env
49 |   env = record {vari = 0 ; varn = c10 }
50 |   proof2 : ((vari env) ≡ 0) ∧ ((varn env) ≡ c10) -- PostCondition
51 |   proof2 = record {pi1 = refl ; pi2 = refl}
52 |
53 | open import Data.Empty
54 | open import Data.Nat.Properties
55 | {-# TERMINATING #-} --
56 |   ↓
57 |   PreCondition(Invalidant)
58 | whileLoop' : {l : Level} {t : Set l} → (env : Env ) → {c10 : ℕ} → ((
59 |   varn env) + (vari env) ≡ c10) → (Code : Env → t) → t
60 | whileLoop' env proof next with ( suc zero ≤? (varn env) )
61 | whileLoop' env proof next | no p = next env
62 | whileLoop' env {c10} proof next | yes p = whileLoop' env1 (proof3 p )
63 |   next
64 |   where
65 |     env1 = record env {varn = (varn env) - 1 ; vari = (vari env) + 1}
66 |     1<0 : 1 ≤ zero → ⊥
67 |     1<0 ()
68 |     proof3 : (suc zero ≤ (varn env)) → varn env1 + vari env1 ≡ c10
69 |     proof3 (s≤s lt) with varn env
70 |     proof3 (s≤s z≤n) | zero = ⊥-elim (1<0 p)
71 |     proof3 (s≤s (z≤n {n'})) | suc n = let open ≡-Reasoning in
72 |       begin
73 |         n' + (vari env + 1)
74 |         ≡⟨ cong ( λ z → n' + z ) ( +-sym {vari env} {1} ) ⟩
75 |         n' + (1 + vari env )

```

```

72 |     ≡⟨ sym ( +-assoc (n') 1 (vari env) ) ⟩
73 |     (n' + 1) + vari env
74 | ≡⟨ cong ( λ z → z + vari env ) +1≡suc ⟩
75 |     (suc n') + vari env
76 | ≡⟨ ⟩
77 |     varn env + vari env
78 | ≡⟨ proof ⟩
79 |     c10
80 |     ■
81 |
82 | -- Condition to Invariant
83 | conversion1 : {l : Level} {t : Set l} → (env : Env) → {c10 : ℕ} →
      ((vari env) ≡ 0) ∧ ((varn env) ≡ c10)
84 |     → (Code : (env1 : Env) → (varn env1 + vari env1 ≡ c10)
      → t) → t
85 | conversion1 env {c10} p1 next = next env proof4
86 |   where
87 |     proof4 : varn env + vari env ≡ c10
88 |     proof4 = let open ≡-Reasoning in
89 |       begin
90 |         varn env + vari env
91 |         ≡⟨ cong ( λ n → n + vari env ) (pi2 p1) ⟩
92 |         c10 + vari env
93 |         ≡⟨ cong ( λ n → c10 + n ) (pi1 p1) ⟩
94 |         c10 + 0
95 |         ≡⟨ +-sym {c10} {0} ⟩
96 |         c10
97 |         ■
98 |
99 | -- all proofs are connected
100 | proofGears : {c10 : ℕ} → Set
101 | proofGears {c10} = whileTest' {} {} {c10} (λ n p1 → conversion1 n p1
      (λ n1 p2 → whileLoop' n1 p2 (λ n2 → (vari n2 ≡ c10))))
102 |
103 | --
104 | -- codeGear with loop step and closed environment
105 | --
106 |
107 | open import Relation.Binary
108 |
109 | record EnvC : Set (succ Zero) where
110 |   field
111 |     c10 : ℕ
112 |     varn : ℕ
113 |     vari : ℕ
114 | open EnvC
115 |
116 | whileTestP : {l : Level} {t : Set l} → (c10 : ℕ) → (Code : EnvC → t)
      → t
117 | whileTestP c10 next = next (record {varn = c10 ; vari = 0 ; c10 = c10})
118 |
119 | whileLoopP : {l : Level} {t : Set l} → EnvC → (next : EnvC → t) → (
      exit : EnvC → t) → t
120 | whileLoopP env next exit with <-cmp 0 (varn env)

```

```

121 | whileLoopP env next exit | tri $\approx$   $\neg$ a b  $\neg$ c = exit env
122 | whileLoopP env next exit | tri $<$  a  $\neg$ b  $\neg$ c =
123 |     next (record env {varn = (varn env) - 1 ; vari = (vari env) + 1 })
124 |
125 | -- equivalent of whileLoopP but it looks like an induction on varn
126 | whileLoopP' : {l : Level} {t : Set l}  $\rightarrow$  (n :  $\mathbb{N}$ )  $\rightarrow$  (env : EnvC)  $\rightarrow$  (n
127 |      $\equiv$  varn env)  $\rightarrow$  (next : EnvC  $\rightarrow$  t)  $\rightarrow$  (exit : EnvC  $\rightarrow$  t)  $\rightarrow$  t
128 | whileLoopP' zero env refl _ exit = exit env
129 | whileLoopP' (suc n) env refl next _ = next (record {c10 = (c10 env) ;
130 |     varn = varn env ; vari = suc (vari env) })
131 |
132 | -- normal loop without termination
133 | {-# TERMINATING #-}
134 | loopP : {l : Level} {t : Set l}  $\rightarrow$  EnvC  $\rightarrow$  (exit : EnvC  $\rightarrow$  t)  $\rightarrow$  t
135 | loopP env exit = whileLoopP env ( $\lambda$  env  $\rightarrow$  loopP env exit ) exit
136 |
137 | whileTestPCall : (c10 :  $\mathbb{N}$ )  $\rightarrow$  EnvC
138 | whileTestPCall c10 = whileTestP {} {} c10 ( $\lambda$  env  $\rightarrow$  loopP env ( $\lambda$  env
139 |      $\rightarrow$  env))
140 |
141 | --
142 | -- codeGears with states of condition
143 | --
144 | data whileTestState : Set where
145 |     s1 : whileTestState
146 |     s2 : whileTestState
147 |     sf : whileTestState
148 |
149 | whileTestStateP : whileTestState  $\rightarrow$  EnvC  $\rightarrow$  Set
150 | whileTestStateP s1 env = (vari env  $\equiv$  0)  $\wedge$  (varn env  $\equiv$  c10 env)
151 | whileTestStateP s2 env = (varn env + vari env  $\equiv$  c10 env)
152 | whileTestStateP sf env = (vari env  $\equiv$  c10 env)
153 |
154 | whileTestPwP : {l : Level} {t : Set l}  $\rightarrow$  (c10 :  $\mathbb{N}$ )  $\rightarrow$  ((env : EnvC)  $\rightarrow$ 
155 |     whileTestStateP s1 env  $\rightarrow$  t)  $\rightarrow$  t
156 | whileTestPwP c10 next = next env record { pi1 = refl ; pi2 = refl } where
157 |     env : EnvC
158 |     env = whileTestP c10 ( $\lambda$  env  $\rightarrow$  env )
159 |
160 | whileLoopPwP : {l : Level} {t : Set l}  $\rightarrow$  (env : EnvC)  $\rightarrow$ 
161 |     whileTestStateP s2 env
162 |      $\rightarrow$  (next : (env : EnvC)  $\rightarrow$  whileTestStateP s2 env  $\rightarrow$  t)
163 |      $\rightarrow$  (exit : (env : EnvC)  $\rightarrow$  whileTestStateP sf env  $\rightarrow$  t)  $\rightarrow$  t
164 | whileLoopPwP env s next exit with  $\leftarrow$ cmp 0 (varn env)
165 | whileLoopPwP env s next exit | tri $\approx$   $\neg$ a b  $\neg$ c = exit env (lem (sym b) s)
166 |     where
167 |         lem : (varn env  $\equiv$  0)  $\rightarrow$  (varn env + vari env  $\equiv$  c10 env)  $\rightarrow$  vari env
168 |              $\equiv$  c10 env
169 |         lem refl refl = refl
170 | whileLoopPwP env s next exit | tri $<$  a  $\neg$ b  $\neg$ c = next (record env {varn = (
171 |     varn env) - 1 ; vari = (vari env) + 1 }) (proof5 a)
172 |     where
173 |         1<0 : 1  $\leq$  zero  $\rightarrow$   $\perp$ 
174 |         1<0 ()

```

```

168   proof5 : (suc zero ≤ (varn env)) → (varn env - 1) + (vari env +
169   1) ≡ c10 env
169   proof5 (s ≤ s lt) with varn env
170   proof5 (s ≤ s z ≤ n) | zero = ⊥-elim (1 < 0 a)
171   proof5 (s ≤ s (z ≤ n {n'})) | suc n = let open ≡-Reasoning in
172   begin
173     n' + (vari env + 1)
174     ≡⟨ cong ( λ z → n' + z ) ( +-sym {vari env} {1} ) ⟩
175     n' + (1 + vari env)
176     ≡⟨ sym ( +-assoc (n') 1 (vari env) ) ⟩
177     (n' + 1) + vari env
178     ≡⟨ cong ( λ z → z + vari env ) +1≡suc ⟩
179     (suc n' ) + vari env
180     ≡⟨ ⟩
181     varn env + vari env
182     ≡⟨ s ⟩
183     c10 env
184     ■
185
186
187   whileLoopPwP' : {l : Level} {t : Set l} → (n : ℕ) → (env : EnvC ) → (n
188     ≡ varn env) → whileTestStateP s2 env
189     → (next : (env : EnvC ) → (pred n ≡ varn env) → whileTestStateP s2
190       env → t)
191     → (exit : (env : EnvC ) → whileTestStateP sf env → t) → t
192   whileLoopPwP' zero env refl refl next exit = exit env refl
193   whileLoopPwP' (suc n) env refl refl next exit = next (record env {varn =
194     pred (varn env) ; vari = suc (vari env) }) refl (+-suc n (vari env))
195
196   {-# TERMINATING #-}
197   loopPwP : {l : Level} {t : Set l} → (env : EnvC ) → whileTestStateP s2
198     env → (exit : (env : EnvC ) → whileTestStateP sf env → t) → t
199   loopPwP env s exit = whileLoopPwP env s (λ env s → loopPwP env s exit )
200     exit
201
202   loopPwP' : {l : Level} {t : Set l} → (n : ℕ) → (env : EnvC ) → (n ≡
203     varn env) → whileTestStateP s2 env → (exit : (env : EnvC ) →
204     whileTestStateP sf env → t) → t
205   loopPwP' zero env refl refl exit = exit env refl
206   loopPwP' (suc n) env refl refl exit = whileLoopPwP' (suc n) env refl
207     refl (λ env x y → loopPwP' n env x y exit) exit
208
209   loopHelper : (n : ℕ) → (env : EnvC ) → (eq : varn env ≡ n) → (seq :
210     whileTestStateP s2 env) → loopPwP' n env (sym eq) seq λ env_1 x → (
211     vari env_1 ≡ c10 env_1)
212   loopHelper zero env eq refl rewrite eq = refl
213   loopHelper (suc n) env eq refl rewrite eq = loopHelper n (record { c10 =
214     suc (n + vari env) ; varn = n ; vari = suc (vari env) }) refl (+-suc n
215     (vari env))
216
217

```

```

208 |
209 | -- all conditons are correctly connected and required condtion is proved
      | in the continuation
210 | --     use required condition as t in (env → t) → t
211 | --
212 | whileTestPCallwP : (c : ℕ) → Set
213 | whileTestPCallwP c = whileTestPwP {} {} c (λ env s → loopPwP env (
      |   conv env s) (λ env s → vari env ≡ c10 env) ) where
214 |   conv : (env : EnvC) → (vari env ≡ 0) ∧ (varn env ≡ c10 env) → varn
      |     env + vari env ≡ c10 env
215 |   conv e record { pi1 = refl ; pi2 = refl } = +zero
216 |
217 |
218 | whileTestPCallwP' : (c : ℕ) → Set
219 | whileTestPCallwP' c = whileTestPwP {} {} c (λ env s → loopPwP' (varn
      |   env) env refl (conv env s) (λ env s → vari env ≡ c10 env) ) where
220 |   conv : (env : EnvC) → (vari env ≡ 0) ∧ (varn env ≡ c10 env) → varn
      |     env + vari env ≡ c10 env
221 |   conv e record { pi1 = refl ; pi2 = refl } = +zero
222 |
223 | helperCallwP : (c : ℕ) → whileTestPCallwP' c
224 | helperCallwP c = whileTestPwP {} {} c (λ envs s → loopHelper c (record
      |   { c10 = c ; varn = c ; vari = zero }) refl +zero)
225 |
226 | --
227 | -- Using imply relation to make soundness explicit
228 | -- termination is shown by induction on varn
229 | --
230 |
231 | data _implies_ (A B : Set) : Set (succ Zero) where
232 |   proof : (A → B) → A implies B
233 |
234 | whileTestPSem : (c : ℕ) → whileTestP c (λ env → ⊤ implies (
      |   whileTestStateP s1 env) )
235 | whileTestPSem c = proof (λ _ → record { pi1 = refl ; pi2 = refl } )
236 |
237 | whileTestPSemSound : (c : ℕ) (output : EnvC) → output ≡ whileTestP c
      |   (λ e → e) → ⊤ implies ((vari output ≡ 0) ∧ (varn output ≡ c))
238 | whileTestPSemSound c output refl = whileTestPSem c
239 |
240 |
241 | whileConvPSemSound : {l : Level} → (input : EnvC) → (whileTestStateP s1
      |   input) implies (whileTestStateP s2 input)
242 | whileConvPSemSound input = proof λ x → (conv input x) where
243 |   conv : (env : EnvC) → (vari env ≡ 0) ∧ (varn env ≡ c10 env) → varn
      |     env + vari env ≡ c10 env
244 |   conv e record { pi1 = refl ; pi2 = refl } = +zero
245 |
246 | loopPP : (n : ℕ) → (input : EnvC) → (n ≡ varn input) → EnvC
247 | loopPP zero input refl = input
248 | loopPP (suc n) input refl =
249 |   loopPP n (record input { varn = pred (varn input) ; vari = suc (vari
      |     input)}) refl
250 |

```

```

251 whileLoopPSem : {l : Level} {t : Set l} → (input : Eenvc) →
    whileTestStateP s2 input
252 → (next : (output : Eenvc) → (whileTestStateP s2 input) implies (
    whileTestStateP s2 output) → t)
253 → (exit : (output : Eenvc) → (whileTestStateP s2 input) implies (
    whileTestStateP sf output) → t) → t
254 whileLoopPSem env s next exit with varn env | s
255 ... | zero | _ = exit env (proof (λ z → z))
256 ... | (suc varn) | refl = next (record env { varn = varn ; vari = suc (
    vari env) }) (proof λ x → +-suc varn (vari env) )
257
258 loopPPSem : (input output : Eenvc) → output ≡ loopPP (varn input)
    input refl
259 → (whileTestStateP s2 input) → (whileTestStateP s2 input) implies (
    whileTestStateP sf output)
260 loopPPSem input output refl s2p = loopPPSemInduct (varn input) input
    refl refl s2p
261 where
262   lem : (n : ℕ) → (env : Eenvc) → n + suc (vari env) ≡ suc (n + vari
    env)
263   lem n env = +-suc (n) (vari env)
264   loopPPSemInduct : (n : ℕ) → (current : Eenvc) → (eq : n ≡ varn
    current) → (loopeq : output ≡ loopPP n current eq)
265   → (whileTestStateP s2 current) → (whileTestStateP s2 current)
    implies (whileTestStateP sf output)
266   loopPPSemInduct zero current refl loopeq refl rewrite loopeq = proof
    (λ x → refl)
267   loopPPSemInduct (suc n) current refl loopeq refl rewrite (sym (lem n
    current)) =
268     whileLoopPSem current refl
269     (λ output x → loopPPSemInduct n (record { c10 = n + suc (
    vari current) ; varn = n ; vari = suc (vari current) }) refl loopeq
    refl)
270     (λ output x → loopPPSemInduct n (record { c10 = n + suc (
    vari current) ; varn = n ; vari = suc (vari current) }) refl loopeq
    refl)
271
272 whileLoopPSemSound : {l : Level} → (input output : Eenvc)
273 → whileTestStateP s2 input
274 → output ≡ loopPP (varn input) input refl
275 → (whileTestStateP s2 input) implies (whileTestStateP sf output)
276 whileLoopPSemSound {l} input output pre eq = loopPPSem input output eq
    pre

```