GearsOS の Agda による記述と検証

外 間 政 尊[†1]　河 野 真 治[†2]

Gears OS は継続を主とするプログラミング言語 CbC で記述されている。 OS やアプリケーションの信頼性を上げるには仕様を満たしていることを確認する必要がある。確認方法にはモデル検査と証明がある。 ここでは定理証明支援系 Agda を用いた、CbC 言語の証明方法を考える。 CbC は関数呼び出しを用いず goto 文により遷移する。これを継続を用いた関数型プログラムとして記述することができる。 この記述は Agda 上で決まった形を持つ関数として表すことができる。 Gears OS のモジュールシステムは、実装と API を分離することを可能にしている。 このモジュールシステムを Agda 上で記述することができた。 継続は不定の型を返す関数で表されるので、継続に直接要求仕様を Agda の論理式として渡すことができる。 継続には仕様以外にも関数を呼び出すときの前提条件 (pre-condition) を追加することが可能である。これにより、Hoare Logic 的な証明を Agda で記述した CbC に直接載せることが可能になる。 Agda で記述された CbC と実装に用いる CbC は並行した形で存在する。つまり、CbC のモジュールシステムで記述されたプログラムを比較的機械的に Agda で記述された CbC 変換することができる。 本論文では Agda 上での CbC の記述手法を検討し、モジュール化を含めた形で検証を行う。

Masataka HOKAMA [†1] and Shinji KONO [†2]

1. 定理証明系 Agda を用いた GearsOS の検証

動作するソフトウェアは高い信頼性を持つことが望ましい。そのためにはソフトウェアが期待される動作をすることを保証する必要がある。また、ソフトウェアが期待される動作をすることを保証するためには検証を行う必要がある。

当研究室では検証の単位として CodeGear、DataGear という単位を用いてソフトウェアを記述する手法を提案しており、 CodeGear 、 DataGear という単位を用いてプログラミングする言語として Countinuation based C [1] (以下 CbC) を開発している。CbC は C 言語と似た構文を持つ言語である。また、CodeGear、DataGear を用いて信頼性と拡張性をメタレベルで保証する GearsOS [2] を CbC で開発している。

本研究では検証を行うために証明支援系言語 Agda [3] を使用している。Agda では型で証明したい論理式を書き、その型に合った実装を記述することで証明を記述することができる。

†1 琉球大学大学院理工学研究科情報工学専攻
Interdisciplinary Information Engineering, Graduate Sc k533 hool of Engineering and Science, University of the Ryukyus.
†2 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus.

本論文では CodeGear、DataGear での記述を Agda で表現した。また、GearsOS で使われている interface の記述を Agda で表現し、その記述を通して実装の仕様の一部に対して証明を行なった。さらに、 Agda で継続を用いた記述をした際得られた知見を示す。

2. CodeGear、 DataGear

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

Gear では処理やデータ構造が CodeGear、DataGear に閉じている。したがって、DataGear は Agda のデータ構造 (data と record) で表現できる。CodeGear は Agda の CPS (Continuation Passing Style) で関数として表現することができる。

CodeGear は Agda では継続を用いた末尾呼び出し

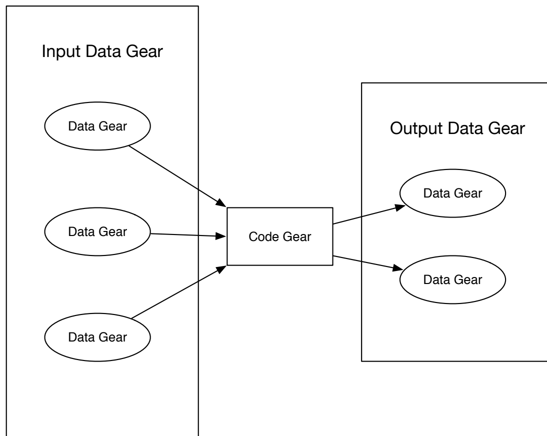

図 1: CodeGear と DataGear の関係

を行う関数として表現される。継続は不定の型 (t) を返す関数で表される。CodeGear 自体も同じ型 t を返す関数となる。例えば、Stack への push を行う関数 pushStack は以下のような型を持つ。

Code 1: pushStack の型

```
1    pushStack : a -> (Stack a si -> t) -> t
```

pushStack が関数名で、コロンの後ろに型を記述する。最初の引数は Stack に格納される型 a を持つ。二つ目の引数は継続であり、Stack a si (si という実装を持つ a を格納する Stack) を受け取り不定の型 t を返す関数である。この CodeGear 自体は不定の型 t を返す。

GearsOS で CodeGear の性質を証明するには、Agda で記述された CodeGear と DataGear に対してメタ計算として証明を行う。証明すべき性質は、不定の型を持つ継続 t に記述することができる。例えば、Stack にある値 x を push して、pop すると x' が取れてくる。Just x と Just x' は等しい必要がある。これは Agda では (Just x ≡ x') と記述される。ここで Just とは Agda の以下のデータ構造である。

Code 2: data 型の例:Maybe

```
1  data Maybe {n : Level } (a : Set n) : Set n
       where
2    Nothing : Maybe a
3    Just    : a -> Maybe a
```

これは DataGear に相当し、Nothing と Just の二つの状態を保つ。pop した時に、 Stack が空であれば Nothing を返し、そうでなければ Just のついた返り値を返す。

この性質を Agda で表すと、以下のような型になる。Agda では証明すべき論理式は型で表される。継続部分に直接証明すべき性質を型として記述できる。Agda ではこの型に対応する λ 項を与えると証明が完了したことになる。

Code 3: push と pop

```
1  push->pop : {l : Level} {D : Set l} (x : D
      )
2          (s : SingleLinkedStack D) ->
3          pushStack (stackInSomeState s)
4          x (\s -> popStack s
5          ( \s3 x1 -> (Just x ≡  x1)))
```

このように、CodeGear を Agda で記述し、継続部分に証明すべき性質を Agda で記述する。

GearsOS での記述は interface によってモジュール化される。よって、このモジュール化も Agda により記述する必要がある。CbC で記述された任意の CodeGear と Meta CodeGear が Agda にそのまま変換されるわけではないが、変換可能なように記述されると仮定する。

以下の節では、Agda の基本について復習を行う。

3. Agda の文法

Agda はインデントに意味を持つため、きちんと揃える必要がある。また、スペースの有無は厳格にチェックされる。

Agda における型指定は : を用いて行う。例えば、変数 x が型 A を持つ、ということを表すには x : A と記述する。

データ型は、代数的なデータ構造で、その定義には data キーワードを用いる。data キーワードの後に data の名前と、型、 where 句を書きインデントを深くした後、値にコンストラクタとその型を列挙する。Maybe(Code 2) はこの data 型の例である。

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

例えば引数が型 A で返り値が型 B の関数は A -> B のように書ける。また、複数の引数を取る関数の型は A -> A -> B のように書ける。この時の型は A -> (A -> B) のように考えられる。前節に出てきた pushStack の型 (Code 1) はこの例である。pushStack の型の本体は Code 4 のようになる。Code 4 では\の表記が出ている。これは λ 式で初めの pushStack で返した stack である s1 を受け取り、次の関数へ渡している。 Agda の λ 式では\の他に λ で表記することもできる。

Code 4: pushStack の関数定義

```
1   pushStack d next = push (stackMethods)
2             (stack) d (\s1 -> next
3             (record {stack = s1
4             ; stackMethods = stackMethods})
     )
```

ここで書かれている record は C における構造体に相当するレコード型というデータを構築しており、 record キーワードの後の の内部に fieldName = value の形で値を列挙していく。複数の値を列挙する際は ; で区切る必要がある。

定義を行う際は record のキーワード後にレコード名、型、 where の後に field キーワードを入れ、フィールド名と型名をを列挙する。record の定義の例として Stack のデータを操作する際に必要なレコード型のデータ Element (Code 5) を例とする。

Element は単方向のリスト構造になっており、datum に格納する任意の型のデータ、next に次の Element 型のデータを持っている。

Code 5: Element の定義

```
record Element {l : Level} (a : Set l) :
    Set l where
  inductive
  constructor cons
  field
    datum : a  -- ‘data‘ is reserved by
    Agda.
    next : Maybe (Element a)
```

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、 popStack の実装である popSingleLinkedStack を使う。

popSingleLinkedStack では stack、 cs の 2 つの引数を取り、with キーワードの後に Maybe 型の top でパターンマッチしている。Maybe 型は nothing と Just のどちらかを返す。そのため、両方のパターンにマッチしている必要がある。パターンマッチの記述では関数名、引数、を列挙して|の後に パターン名 = で挙動を書く場合と、 Code 6 のように、 ... |で関数名、引数を省略してパターン名 =で挙動を書く方法がある。

また、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。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。

Code 6: パターンマッチの例

```
popSingleLinkedStack stack cs with (top
    stack)
...     | Nothing = cs stack  Nothing
...     | Just d  = cs stack1 (Just data1)
  where
    data1  = datum d
    stack1 = record { top = (next d) }
```

popStack は stack を引数として受け取り、stack の top を取って top を次の Element に変更した新しい stack を構築し、継続に stack と data を渡す interface で、実装部分の popSingleLinkedStack に接続されている。

pushStack と popStack を使った証明の例は Code7 のようになる。ここでは、stack に対し push を行なった直後に pop を行うと取れるデータは push したものと同じになるという論理式を型に書き、証明を行なった。

Code 7: push と pop を使った証明

```
push->pop : {l : Level} {D : Set l}
        (x : D) (s : SingleLinkedStack D)
        -> pushStack (stackInSomeState s) x
        (\s1 -> popStack s1 (\s3 x1
        -> (Just x ≡  x1)))
push->pop {l} {D} x s = refl
```

証明の関数部分に出てきた refl は左右の項が等しいことを表す $x \equiv x$ を生成する項であり、$x \equiv x$ を証明したい場合には refl と書く事ができる。

Code 8: reflection の定義

```
data ≡ __ {a} {A : Set a} (x : A) : A →
     Set a where
  refl : x ≡  x
```

また、Code 9 のように継続を用いて記述することで関数の中で計算途中のデータ内部を確認することができた。ここでは λ 式のネストになり見づらいため、() をまとめる糖衣構文$を使っている。$を先頭に書くことで後ろの一行を () でくくることができる。

Code 9 のように記述し、C-c C-n(Compute nomal form) で関数を評価すると最後に返している stack の top を確認することができる。top の中身は Code9 の中にコメントとして記述した。

Code 9: 継続によるテスト

```
testStack08 = pushSingleLinkedStack
    emptySingleLinkedStack 1
   $ \s -> pushSingleLinkedStack s 2
   $ \s -> pushSingleLinkedStack s 3
   $ \s -> pushSingleLinkedStack s 4
   $ \s -> pushSingleLinkedStack s 5
   $ \s -> top s

-- Just (cons 5 (Just (cons 4 (Just (cons 3
     (Just (cons 2 (Just (cons 1 Nothing)))
    ))))))
```

4. Agda での Stack、Binary Tree の実装

ここでは Agda での Stack 、 Binary Tree の実装を示す。

Stack の実装を以下の Code 10 で示す。実装は SingleLinkedStack という名前の record で定義されている。

Code 10: Agda における Stack の実装の一部

```
pushSingleLinkedStack : {n m : Level } {t : Set m } {Data : Set n} → SingleLinkedStack Data → Data → (Code : SingleLinkedStack Data → t) → t
pushSingleLinkedStack stack datum next = next stack1
  where
    element = cons datum (top stack)
    stack1  = record {top = Just element}

-- Basic stack implementations are specifications of a Stack

singleLinkedStackSpec : {n m : Level } {t : Set m } {a : Set n} → StackMethods {n } {m} a {t} (SingleLinkedStack a)
singleLinkedStackSpec = record {
                          push = pushSingleLinkedStack
                        ; pop  = popSingleLinkedStack
                        }

createSingleLinkedStack : {n m : Level } {t : Set m } {a : Set n} → Stack {n} {m} a {t} (SingleLinkedStack a)
createSingleLinkedStack = record {
                         stack = emptySingleLinkedStack ;
                         tackMethods = singleLinkedStackSpec
                         }
```

SingleLinkedStack 型では、この Element の top 部分のみを定義している。

Stack に対する push 操作では stack と push する element 型の datum を受け取り、datum の next に現在の top を入れ、stack の top を受け取った datum に切り替え、新しい stack を返すというような実装をしている。

Tree の実装 (以下の Code 11) は Tree という record で定義されている。

Code 11: Agda における Tree の実装

```
record TreeMethods {n m : Level } {a : Set n } {t : Set m } (treeImpl : Set n ) : Set (m Level.⊔ n) where
  field
    putImpl : treeImpl → a → (treeImpl → t) → t
    getImpl  : treeImpl → (treeImpl → Maybe a → t) → t

record Tree  {n m : Level } {a : Set n } {t : Set m } (treeImpl : Set n ) : Set (m Level.⊔ n) where
  field
    tree : treeImpl
    treeMethods : TreeMethods {n} {m} {a} {t} treeImpl
  putTree : a → (Tree treeImpl → t) → t
  putTree d next = putImpl (treeMethods ) tree d (\t1 → next (record {tree = t1 ; treeMethods = treeMethods} ))
  getTree : (Tree treeImpl → Maybe a → t) → t
  getTree next = getImpl (treeMethods ) tree (\t1 d → next (record {tree = t1 ; treeMethods = treeMethods} ) d )
```

Tree を構成する Node の型は `Node` 型で定義され key、 value、 Color、rihgt、left などの情報を持っている。Tree を構成する末端の Node は `leafNode` 型で定義されている。

Tree 型の実装では root の Node 情報と Tree に関する計算をする際に、そこまでの Node の経路情報を保持するための Stack を持っている。

Tree の put 操作では tree 、put するノードのキーと値 (k1、value) を引数として受け取り、Tree の root に Node が存在しているかどうかで場合分けしている。Nothing が返ってきたときは RedBlackTree 型の tree 内に定義されている root に受け取ったキーと値を新しいノードとして追加する。Just が返ってきたときは root が存在するので、経路情報を積むために nodeStack を初期化し、受け取ったキーと値で新たなノードを作成した後、ノードが追加されるべき位置までキーの値を比べて新しい Tree を返すというような実装になっている。

5. Agda における Interface の実装

Agda で GearsOS のモジュール化の仕組みである interface を実装した。interface とは、実装と仕様を分ける記述でここでは Stack の実装を SingleLinked-Stack 、 Stack の仕様を Stack とした。interface は record で列挙し、Code 12 のように紐付けることができる。Agda では型を明記する必要があるため record 内に型を記述している。

例として Agda で実装した Stack 上の interface (Code 12) の一部を見る。Stack の実装は SingleLinkedStack として書かれている。それを Stack 側から interface を通して呼び出している。

ここでの interface の型は Stack の record 内にある pushStack や popStack などで、実際に使われる Stack の操作は StackMethods にある push や pop である。この push や pop は SingleLinkedStack で実装されている。

Code 12: Agda における Interface の定義の一部

```
record StackMethods {n m : Level } (a : Set n ) {t : Set m }(stackImpl : Set n ) : Set (m Level.⊔ n) where
  field
    push : stackImpl → a → (stackImpl → t) → t
    pop  : stackImpl → (stackImpl → Maybe a → t) → t
open StackMethods

record Stack {n m : Level } (a : Set n ) {t : Set m } (si : Set n ) : Set (m Level.⊔ n) where
  field
    stack : si
```

```
10      stackMethods : StackMethods {n} {m} a {
        t} si
11    pushStack :  a → (Stack a si → t) → t
12    pushStack d next = push (stackMethods ) (
        stack ) d (\s1 → next (record {stack =
         s1 ; stackMethods = stackMethods } ))
13    popStack : (Stack a si → Maybe a  → t)
        → t
14    popStack next = pop (stackMethods ) (
        stack ) (\s1 d1 → next (record {stack
        = s1 ; stackMethods = stackMethods })
        d1 )
15  open Stack
```

interface を通すことで、実際には Stack の push では stackImpl と何らかのデータ a を取り、 stack を変更し、継続を返す型であったのが、pushStack では 何らかのデータ a を取り stack を変更して継続を返す型に変わっている。

また、 Tree でも interface を記述した。

Code 13: Tree Interface の定義

```
1  record TreeMethods {n m : Level } {a : Set
       n } {t : Set m } (treeImpl : Set n ) :
       Set (m Level.⊔ n) where
2    field
3      putImpl : treeImpl → a → (treeImpl
       → t) → t
4      getImpl  : treeImpl → (treeImpl →
       Maybe a → t) → t
5  open TreeMethods
6
7  record Tree  {n m : Level } {a : Set n } {t
        : Set m } (treeImpl : Set n ) : Set (m
        Level.⊔ n) where
8    field
9      tree : treeImpl
10     treeMethods : TreeMethods {n} {m} {a} {
       t} treeImpl
11   putTree : a → (Tree treeImpl → t) → t
12   putTree d next = putImpl (treeMethods )
       tree d (\t1 → next (record {tree = t1
       ; treeMethods = treeMethods} ))
13   getTree : (Tree treeImpl → Maybe a → t)
        → t
14   getTree next = getImpl (treeMethods )
       tree (\t1 d → next (record {tree = t1
       ; treeMethods = treeMethods} ) d )
15 open Tree
16
17 record RedBlackTree {n m : Level } {t : Set
        m} (a k : Set n) : Set (m Level.⊔ n)
       where
18   field
19     root : Maybe (Node a k)
20     nodeStack : SingleLinkedStack  (Node a
       k)
21     compare : k → k → CompareResult {n}
22 open RedBlackTree
```

interface を記述することによって、データを push する際に予め決まっている引数を省略することができた。また、Agda で interface を記述することで CbC 側では意識していなかった型が、明確化された。

6. Agda による Interface 部分を含めた Stack の部分的な証明

Stack の Interface を使い、 Agda で Interface を経由した証明を行なった。ここでの証明とは Stack の処理が特定の性質を持つことを保証することである。

Stack の処理として様々な性質が存在する。例えば、

- Stack に push した値は存在する
- Stack に push した値は取り出すことができる
- Stack に push した値を pop した時、その値は Stack から消える
- どのような状態の Stack に値を push しても中に入っているデータの順序は変わらない
- どのような状態の Stack でも、値を push した後 pop した値は直前に入れた値と一致する

などの性質がある。

ここでは「どのような状態の Stack でも、値を push した後 pop した値は直前に入れた値と一致する」という性質を証明する。

まず始めに不定状態の Stack を定義する。 Code 14 の stackInSomeState 型は引数として `SingleLinkedStack` 型の `s` を受け取り新しいレコードを返す関数である。この関数により、中身の分からない抽象的な Stack を表現している。ソースコード 14 の証明ではこの `stackInSomeState` に対して、`push` 操作を 2 回行い、`pop` を 2 回行なって取れたデータは push したデータと同じものになることを証明している。

この証明では stackInSomeState 型の s が抽象的な Stack で、そこに x 、 y の 2 つのデータを push している。また、 pop2 で取れたデータは y1 、 x1 となっていて両方が Just で返ってくるかつ、 x と x1 、 y と y1 がそれぞれ合同であることが仮定として型に書かれている。

この関数本体で返ってくる値は $x \equiv x1$ と $y \equiv y1$ のため record でまとめて refl で推論が通る。これにより、抽象化した Stack に対して push 、 pop を行うと push したものと同じものを受け取れることが証明できた。

Code 14: 抽象的な Stack の定義と push→push→pop2 の証明

```
1 stackInSomeState : {l m : Level} {D : Set l
      } {t : Set m} (s : SingleLinkedStack D)
       → Stack {l} {m} D {t}  (
      SingleLinkedStack  D)
2 stackInSomeState s =  record { stack = s ;
      stackMethods = singleLinkedStackSpec }
3
4 push→push→pop2 : {l : Level} {D : Set l}
      (x y : D) (s : SingleLinkedStack D) →
      pushStack (stackInSomeState s) x (\s1
      → pushStack s1 y (\s2 → pop2Stack s2
      (\s3 y1 x1 → (Just x ≡ x1) ∧
       (Just y ≡ y1))))
5 push→push→pop2 {l} {D} x y s = record {
```

```
    pi1 = refl ; pi2 = refl}
```

7. ま と め

本論文では、Agda を用いて GearsOS 上のモジュール化である interface の記述について検討、実装した。また、継続を用いた記述をすることで計算途中のデータの形などを確認することができることが分かった。その他に、interface の記述を通しての証明が行えることが分かった。

今後は Agda での継続に hoare logic をベースにした検証方法を考案、実装し、実際のプログラムに対して検証が可能かどうかを検討する。

参 考 文 献

1) TOKKMORI, K. and KONO, S.: Implementing Continuation based language in LLVM and Clang, *LOLA 2015* (2015).
2) 宮城 光希, 河野 真治: CbC 言語による OS 記述 (2017).
3) : The Agda wiki, `http://wiki.portal.chalmers.se/agda/pmwiki.php`. Accessed: 2018/4/23(Fri).
4) 河野真治，伊波立樹，東恩納琢偉：Code Gear、Data Gear に基づく OS のプロトタイプ，情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2016).
5) 健太比嘉，真治河野：Verification Method of Programs Using Continuation based C, 情報処理学会論文誌プログラミング（PRO）， Vol.10, No.2, pp.5–5 (2017).
6) Norell, U.: Towards a practical programming language based on dependent type theory, PhD Thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (2007).
7) Ek, L., Holmström, O. and Andjelkovic, S.: Formalizing Arne Andersson trees and left-leaning Red-Black trees in Agda.
8) : Welcome to Agda’s documentation! — Agda latest documentation，`http://agda.readthedocs.io/en/latest/`. Accessed: 2018/4/23(Mon).
9) Stump, A.: *Verified Functional Programming in Agda*, Association for Computing Machinery and Morgan & Claypool, New York, NY, USA (2016).