Gears Agda 上のモデル検査の形式化

上地 悠斗[1,a)]　河野 真治[2,b)]

概要：当研究室では Continuation based C （CbC） という言語を用いて，拡張性と信頼性を両立する OS である Gears OS を開発している．CbC とは，C 言語から通常の関数呼び出しではなく，アセンブラでいう jmp 命令により関数を遷移する継続を導入した C 言語の下位言語である．すでに Gears OS はメタ計算によるモデル検査機構を持っている．GearsOS の CodeGear/DataGear は Gears Agda により形式証明に向いた形に記述することができる．モデル検査機構を Gears Agda により記述することで Hoare Logic 的な逐次実行型のプログラムの証明だけでなく，並行実行を含むプログラムの証明が可能になる．

1. Gears Agda でのモデル検査

思い思いにプログラムを書くと，冗長なコードができてしまい，実行時間も遅い場合がある．この場合にコードに対してアルゴリズムを適応すると実行が最適化され実行時間が減り，かつ第三者がコードを読んだ際にロジックが統一されているため理解が容易くなる．つまり，一般的に良いコードが作成できる．しかし，世の中にはすでに大量のアルゴリズムが存在するため，これを一人のプログラマーが全て覚え，適応できる場面を思いつくというのは不可能に近い．その道に詳しい人が複数人いる場面というのも稀だと考えられる．そのため，人が書いたコードに対してアルゴリズムを使用するコードに変換できるようにしたい．この際，アルゴリズム適応前後で処理が変わっていないか検証するのは普通のプログラミング言語では難しい．一般的なプログラミング言語では，関数の遷移が自由であることから，関数遷移などで発生した暗黙の環境が存在するためである．

この問題を解決するため，Gears Agda を用いる．Gears Agda とは当研究室で開発している Continuation based C （CbC） の概念を取り入れた記法で書かれた Agda のことで，通常のプログラミング言語では関数を実行する際にはメインルーチンからサブルーチンに遷移する．この際メインルーチンで使用していた変数などの環境はスタックされ，サブルーチンが終了した際にメインルーチンに戻り，スタックしていた変数をもとに戻す流れとなる．CbC の場合はサブルーチンコールの際にアセンブラで言う jmp で関数遷移を行うことができ，スタックを持たず環境を保持しない．更に遷移後にメインルーチンに戻ることもない．つまり関数の実行では暗黙な環境が存在せず，関数が受け取った引数のみを使用する．これにより限定的な実行条件を作成でき，検証がしやすくなる．

現在，アルゴリズムの適応可否は以下の方法を考えている．あらかじめ，アルゴリズムの実装と検証をおこなったアルゴリズム Sets を用意しておく．書いたコードが事前に定義していたアルゴリズムの仕様を満たしているかを検証していき，満たしているコードがあった場合にそのコードを事前に定義してあるアルゴリズムに入れ替える方針を考えている．この際，実装が仕様を満たしているか検証する手法には，定理証明やモデル検査などが挙げられる．アルゴリズムの入れ替え可否判定には Gears Agda でモデル検査を行い，アルゴリズムの仕様がコードに適応できるか検証するのが妥当だと考えている．思い思いに書いたコードに対して定理証明を行うのはコストが高く，適応するものの内部動作が一致しない場合定理証明を行っても使えないためである．

本論文では Gears Agda でのモデル検査の先駆けとして Dining philosophers problem （DPP） のモデル検査を行う．

2. Continuation based C

Continuation based C [2]（以下 CbC）は関数呼び出しの際に jmp 命令で遷移をし，環境を持たずに遷移することができる C 言語である，すなわち C 言語の下位言語にあたり，よりアセンブラに近い記述を行う．

CbC では 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．

1　琉球大学大学院理工学研究科工学専攻知能情報プログラム
2　琉球大学大学工学部工学科知能情報コース
a)　soto@cr.ie.u-ryukyu.ac.jp
b)　kono@ie.u-ryukyu.ac.jp

jmp 命令で関数遷移するため関数遷移し実行が終了しても，もとの関数に戻ることはない．そのため次に遷移する Code Gear を指定する．これは，関数型プログラミングでの末尾関数呼び出しに相当する．したがって，Code Gear に Deta Gear を与え，それをもとに処理を行い，出力として Data Gear を返し，また次の Code Gear に遷移していく流れとなる．

他のプログラミング言語とは違い，Code Gear が 暗黙の環境を持たず，受け取った Data Gear のみをもとに処理をすること，さらに Code Gear 単位で処理が完結していることから，検証に適したプログラミング言語であると言える．

また，プログラムを記述する際は，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．

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 1 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．

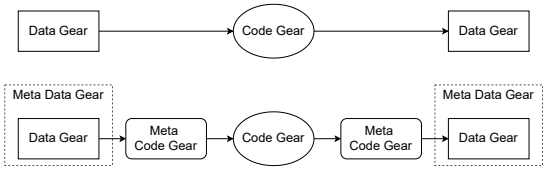

図 1: メタ計算を可視化した CodeGear と DataGear

3. GearsAgda 形式で書く Agda

CbC の継続の概念を取り入れた Agda の記法を説明する．Agda では関数の再帰呼び出しが可能であるが，CbC では値が 帰って来ない．そのため Agda で実装を行う際には再帰呼び出しを行わないようにする．

以下で Gears Agda の記述方法を足し算を行うプログラムを用いて説明する．

```
record Env : Set where
  field
    varx : ℕ
    vary : ℕ
open Env
```

Code 1: Agda での Data Gear の定義

```
plus-c : {l : Level} {t : Set l} → Env → (exit :
    Env → t) → t
plus-c env exit = plus-p (vary env) env exit where
  plus-p : {l : Level} {t : Set l} → ℕ → Env → (
      exit : Env → t) → t
  plus-p zero env exit = exit env
  plus-p (suc reducer) env exit = plus-p reducer
      record env{varx = (suc (varx env)) ; vary =
      reducer} exit
```

Code 2: Agda での Code Gear の定義

```
{-# TERMINATING #-}
plus-c-term : {l : Level} {t : Set l} → Env → (exit
    : Env → t) → t
plus-c-term env exit with vary env
... | zero = exit (record { varx = varx env ; vary =
    vary env })
... | suc y = plus-c-term (record { varx = suc (varx
    env) ; vary = y }) exit
```

Code 3: Agda での 停止性が示せない CodeGear の例

```
plus : ℕ → ℕ → Env
plus x y = plus-c (record { varx = x ; vary = y }) (
    λ env → env)
```

Code 4: Agda での CodeGear の初期化

Code 1 が Data Gear の定義をしている．今回は足し算を実装するので，varx に vary を足すことを考える．そのためそれらが 2 つの自然数を持つようにしている．

Code 2 では Code Gear の定義になる．最初に Data Gear となる env を受け取ったあと，そのまま次の関数に遷移させている．

Agda の記述は Curry-Howard 対応になっていて，最初に関数名のあとに:（コロン）の後ろに命題を記述し，そのあとに関数名のあとに引数を書き，=（イコール）の後ろに定義を記述しています．

Gears Agda での Code Gear の命題は必ず （Env → t） → t で終了するようになっている．この （Env → t） は引数で受け取る型で Env を受け取って t を返すという意味になる．これが Code Gear を実行したあとの末尾関数呼び出しを行う次の Code Gear となる．最後に t を返すとなっているのは，これ自体が Code Gear であることを示している．

受け取ったあとに別の関数に再度渡している．これは後述するが，Agda の繰り返し処理を行う際に停止性を見失うために減少列を引数に取っている．内部の処理は reducer を減らしながら varx を増やし，vary の値を varx に与えていくことで足し算を定義している．基本的に繰り返し実行するコードを実装する場合には，実行時に減少しその関数がいずれ停止することを示す reducer を含めるようにしている．

reducer を含めなかった際の Code Gear を Code 3 に示す．Agda ではパターンマッチを行うことで場合分けを考

えることができるが，受け取った Code Gear である env を with を使用してパターンマッチを試みている．パターンマッチ自体は可能だが，この方法だと Agda が関数が停止することを認識できない．そのため，{-# TERMINATING #-} を関数定義の前にアノテーションしこの関数が停止することを記述してコンパイルが通るようにしている．

Code 4 は受け取った引数で Data Gear を初期化してそれを Code Gear に与えることで実行を行っている．今回の例では 引数から Data Gear を作成するのは複雑ではないため，一度で Data Gear を作成してそれを Code Gear に渡している引数から Data Gear を作成するのが複雑な場合は一度 入力から Data Gear を作成する Code Gear を用いる．加えて，実行なので命題の部分の最後が Env になっている．

3.1 Agda による Meta Gears

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．今回はモデル検査を行う際に使用する

- Meta DataGear
  Agda 上で 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．通常の Data Gear を wraping している．今回はこれを用いることで，モデル検査の状態を保存する
- Meta CodeGear
  Meta CodeGear は 通常の CodeGear では扱えないメタレベルの計算を扱う CodeGear である．Agda での Meta CodeGear は Meta DataGear を引数に取りそれらの関係を返す CodeGear である．今回はここでモデル検査を行う

4. モデル検査

モデル検査とは，検証手法の一つである．他の検証手法として代表的なものとして，定理証明が挙げられる．

モデル検査はプログラムが入力に対して仕様を満たした動作を行うことを網羅的に検証することを指す．

モデル検査と定理証明を比較した際に，モデル検査は入力が無限になる状態爆発が起こり得るという問題がある．定理証明では数学的な証明をするため状態爆発を起こさず検証を行うことができるが，専門的な知識が多く難易度が高いという欠点がある．

5. Dining Philosophers Problem

今回はモデル検査を行う対象として Dining Philosophers Problem（以下 DPP）を用いることとした．DPP とは資源共有問題であり，モデル検査をする際に挙げられる代表的な問題である．

DPP のストーリーを図 2 に示している．

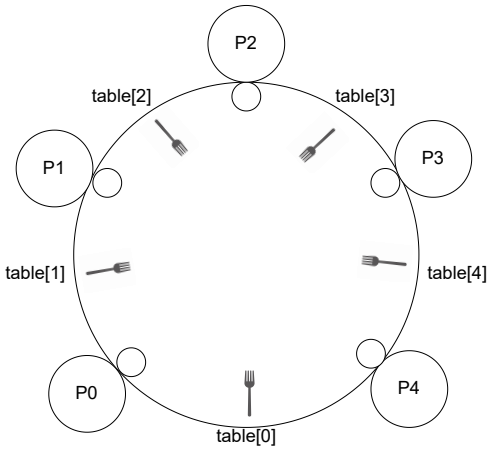

図 2: メタ計算を可視化した CodeGear と DataGear

したがって，哲学者は以下のようなフローを独立して並列に繰り返し実行することとなる．

(1) しばらくの間思考を行う
(2) 食事をするために右のフォークを取る
(3) 右のフォークを取ったら，次は左のフォークを取る
(4) 両方のフォークを取ったら，しばらく食事をする
(5) 思考に戻るために左手に持っているフォークをテーブルに置く
(6) 左手のフォークを置いたあとは右のフォークをテーブルに置く

この際，すべての哲学者が同時に右のフォークを取った場合のことを考える．すべての哲学者はフォークを持っている．次に哲学者は左のフォークを取ろうとする．しかしフォークは哲学者の人数と同じ数だけ存在しているため，テーブルの上にフォークはすでにひとつも存在しない．すべての哲学者は左のフォークを取ろうとするが誰も右のフォークを置くことがないため，すべての哲学者の動作がこの状態で止まる．（dead lock）これが起こることを Gears Agda で検出したい．

5.1 Gears Agda による DPP の実装

DPP の記述の主要部分を示し，説明する．

```
data Code : Set where
  C_putdown_rfork : Code
  C_putdown_lfork : Code
  C_thinking : Code
  C_pickup_rfork : Code
  C_pickup_lfork : Code
  C_eating : Code
```

Code 5: Gears Agda での DPP の 哲学者の状態

```
record Phi : Set where
  field
    pid : ℕ
```

```
    right-hand : Bool
    left-hand : Bool
    next-code : Code
open Phi
```

Code 6: Gears Agda での DPP の プロセス

```
record Env : Set where
  field
    table : List ℕ
    ph : List Phi
open Env
```

Code 7: Gears Agda での DPP の Data Gear

Code 5 は前述した哲学者の状態を書き記して，哲学者が今行おうとしている動作を網羅している．

Code 6 は哲学者一人ずつの環境を持っている．pid はその哲学者がどこに座っているかの識別子で，right / left hand はフォークを手に持っているかを格納している．next-code は次に行う動作を格納している．

Code 7 が Data Gear になる．ph は前もって定義した一人の哲学者のプロセスの List になる．List になっている理由は，哲学者が複数人いるためである．そのため実行時に List から一人ずつ取り出して実行をしていく．

table はテーブルに置いてあるフォークの状態のことで，pid が 1 の人の右側にあるフォークが List の最初にあり，pid が 1 の人の左側にあるフォーク，すなわち pid が 2 の人の右側にあるフォークがその次の List に格納されていくようになっている．また，自然数の List になっているが，その場所のフォークがテーブルの上にある場合は自然数の 0 が，誰かが所持している場合はその人の pid が格納されるようになっている．

```
init-table : {n : Level} {t : Set n} → ℕ → (exit :
    Env → t) → t
init-table n exit = init-table-loop n 0 (record {
    table = [] ; ph = []}) exit where
  init-table-loop : {n : Level} {t : Set n} → (redu
      inc : ℕ) → Env → (exit : Env → t) → t
  init-table-loop zero ind env exit = exit env
  init-table-loop (suc redu) ind env exit = init-
      table-loop redu (suc ind) record env{
    table = 0 :: (table env)
    ; ph = record {pid = redu ; left-hand = false ;
        right-hand = false ; next-code = C_thinking }
      :: (ph env) } exit
```

Code 8: Gears Agda での DPP の Data Gear の init

Code 8 が入力から Data Gear を作成する Code Gear になる．ここでは哲学者の人数を自然数で受け取り，人数分の List Phi と table を一つづつ作成し env を作成している．また，最初の哲学者の状態は思考することであるため，next-code には C_thinking を格納している．

```
code_table : {n : Level} {t : Set n} → Code → ℕ →
    Phi → Env → (Env → t) → t
code_table C_putdown_rfork = putdown-rfork-c
code_table C_putdown_lfork = putdown-lfork-c
code_table C_thinking = thinking-c
code_table C_pickup_rfork = pickup-rfork-c
code_table C_pickup_lfork = pickup-lfork-c
code_table C_eating = thinking-c
```

Code 9: Gears Agda での DPP の step 実行

Agda では並列実行を行うことができない．そのため step 単位の実行を一つづつ行うことで並列実行をしていることとする．

この際に Env にある List Phi の中身を展開しながら一つづつ行動を処理していく．

```
pickup-lfork-c : {n : Level} {t : Set n} → ℕ → Phi
    → Env → (Env → t) → t
pickup-lfork-c ind p env exit = pickup-lfork-p (suc
    ind) [] (table env) p env exit where
  pickup-lfork-p : {n : Level} {t : Set n} → ℕ → (f
      b : List ℕ) → Phi → Env → (Env → t) → t
  pickup-lfork-p zero f [] p env exit with table env
  ... | [] = exit env
  ... | 0 :: ts = exit record env{ph = ((ph env) ++ (
      record p{left-hand = true ; next-code =
      C_eating} :: [])); table = ((pid p) :: ts)}
  ... | (suc x) :: ts = exit record env{ph = ((ph env
      ) ++ p :: [])}
  pickup-lfork-p zero f (0 :: ts) p env exit = exit
      record env{ph = ((ph env) ++ (record p{left-
      hand = true ; next-code = C_eating} :: []));
      table = (f ++ ((pid p) :: ts))}
  pickup-lfork-p zero f ((suc x) :: ts) p env exit =
      exit record env{ph = ((ph env) ++ p :: [])}
  pickup-lfork-p (suc ind) f [] p env exit = exit env
  pickup-lfork-p (suc ind) f (x :: ts) p env exit =
      pickup-lfork-p ind (f ++ (x :: [])) ts p env
      exit
```

Code 10: Gears Agda での DPP の左のフォークを取る記述

Code 10 が step 実行をした際に哲学者が左側のフォークを取る記述になる．

左側のフォークを取る際には table の index は pid の次の値になっている．図 2 を見ると直感的に理解ができる．

最後の哲学者は一番最初のフォークを参照する必要がある．フォークの状態を確認し，値が 0 である場合はフォークがテーブルの上にあるのでそれを自分の pid に書き換え left-hand を true に書き換えてフォークを手に持つフォークの状態が 0 以外，すなわち他の哲学者がその場所のフォークを取得している場合は状態を変化させずに処理を続ける．このように左のフォークを取る記述をした．

右側のフォークを取る場合は引数の部分を 1 足さずにそのまま受け取る．比較するべき table の List と哲学者の List は一致しているため，put_lfork のように最後の哲学者が最初のフォークを参照することもない．

似たような形で哲学者がフォークを置く putdown-l/rfork を実装した．

思考と食事の実装に関してはそのまま状態を変更することなく進むようにしている．

6. DPP のモデル検査

モデル検査の機能として，入力の網羅が挙げられる．今回の DPP の入力の網羅として，哲学者が思考をつづけるのか，食事をはじめようとするのかと食事中に食事をそのままつづけるのか，思考をするために食事を止めようとするのかが分岐する．

そのため， next-code が thinking か eating であるものに対して分岐を網羅するコードを実装した．

内部で行っていることとして，その Code Gear 内に存在している next-code が thinking もしくは eating である場合にそのプロセスの next-code をそのままにするか，それぞれ pickup-rfork か putdown-lfork にする．そのため，その部分に対して bit 全探索を行い，場合の網羅を行っている．

7. まとめと今後の課題

今回は Agda に CbC の継続の概念を追加した Gears Agda にて DPP のモデル検査を行おうとした．結果として，DPP の実装と入力の網羅までできた．

今後はプロセスがすべてほかのプロセスの終了待ちになった場合に dead lock 状態になっていることを検知できるようにしたい．加えて，assert の機能をつけて仕様通りの動作がされていることを検証したい．

参考文献

[1] : The Agda wiki, `http://wiki.portal.chalmers.se/agda/pmwiki.php`.
[2] Kaito, T. and Shinji, K.: Implementing Continuation based language in LLVM and Clang, *LOLA 2015, Kyoto* (2015).
[3] Stump, A.: *Verified Functional Programming in Agda*, Association for Computing Machinery and Morgan & Claypool, New York, NY, USA (2016).
[4] 伊波立樹：Gears OS の並列処理，修士論文，琉球大学 大学院理工学研究科情報工学専攻 (2018).
[5] 政尊外間，真治河野：GearsOS の Agda による記述と検証，技術報告 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 (2018).
[6] 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 (online), available from ⟨https://ci.nii.ac.jp/naid/170000148438/en/⟩ (2017).
[7] 徳森海斗：LLVM Clang 上の Continuation based C コンパイラの改良，修士論文，琉球大学 大学院理工学研究科情報工学専攻 (2016).
[8] 比嘉健太：メタ計算を用いた Continuation based C の検証手法，修士論文，琉球大学大学院理工学研究科 情報工学専攻 (2017).