GearsAgda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 GearsOS 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 GearsOS 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 GearsAgda 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 GearsAgda 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. プログラミング言語の検証

OS やアプリケーションの信頼性を高めることは重要な課題である。信頼性を高める為にはプログラムが仕様を満たした実装を検証する必要がある。具体的には「モデル検査」や「定理証明」などが検証手法としてあげられる。

当研究室では Continuation based C (CbC) という言語を開発している。CbC とは、C 言語からループ制御構造とサブルーチンコールを取り除き、継続を導入した C 言語の下位言語である。その為、それを実装した際のプログラムが正確に動作するのか検証を行いたい。

仕様に合った実装を実施していることの検証手法として 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でコマンドの検証を行う。この定義が CbC の実行を継続するという性質と相性が良い。

CbC では実行を継続するため、ある関数の実行結果は事後条件になるが、その実行結果が遷移する次の関数の事前条件になる。それを繋げていくため、個々の関数の正当性を証明することと接続の健全性について証明するだけでプログラム全体の検証を行うことができる。

CbC ではループ制御構造を取り除いているため、CbC にてループが含まれるプログラムを作成した際の検証を行う必要がある。先行研究では CbC における WhileLoop の検証を行なっている。

Agda が変数への再代入を許していない為、ループが存在し、かつ再代入がプログラムに含まれる RedBlackTree の検証を行いたい。

これらのことから、CbC に対応するように Agda で RedBlackTree を記述し、Hoare Logic により検証を行うことを目指す。

2. Continuation based C

Continuation based C [15] (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 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 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。

また、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

[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

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。

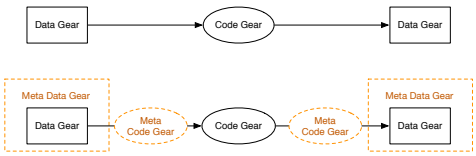

図 1: メタ計算を可視化した CodeGear と DataGear

Agda [19] は純粋関数型言語である。Agda は依存型という型システムを持ち、型を第一級オブジェクトとして扱う。

Agda の記述ではインデントが意味を持ち、スペースの有無もチェックされる。コメントは `-- comment` か `{-- comment --}` のように記述される。また、_でそこに入りうるすべての値を示すことができ、?でそこに入る値や型を不明瞭なままにしておくことができる。

Agda では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。

Code 1 は自然数の型である ℕ (Natural Number) を例である。

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

Listing 1: 自然数を表すデータ型 Nat の定義

Nat では zero と suc の 2 つのコンストラクタを持つデータ型である。suc は ℕ を受け取って ℕ を表す再帰的なデータになっており、suc を連ねることで自然数全体を表現することができる。

ℕ 自身の型は Set であり、これは Agda が組み込みで持つ「型集合の型」である。Set は階層構造を持ち、型集合の集合の型を指定するには Set1 と書く。

Agda には C 言語における構造体に相当するレコード型というデータも存在する、例えば x と y の二つの自然数からなるレコード Point を定義する。Code 2 のようになる。

```
record Envc : Set where
  field
    vari : ℕ
    varn : ℕ
    c10 : ℕ

makeEnv : ℕ → ℕ → ℕ → Envc
makeEnv i n c = record { vari = i ; varn = n ; c10 =
    c }
```

Listing 2: Agda におけるレコード型の定義

レコードを構築する際は record キーワード後の {} の内部に FieldName = value の形で値を列挙する。複数の値を列挙するには ; で区切る必要がある。

Agda での関数は型の定義と、関数の定義をする必要がある。関数の型はデータと同様に : を用いて name : type に記述するが、入力を受け取り出力返す型として記述される。→ 、 または→ を用いて input → output のように記述される。また、_+_のように関数名で_を使用すると引数がその位置にあることを意味し、中間記法で関数を定義することもできる。関数の定義は型の定義より下の行に、= を使い name input = output のように記述される。

例えば引数が型 A で返り値が型 B の関数は A → B のように書くことができる。また、複数の引数を取る関数の型は A → A → B のように書ける。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 Code 3 のように定義できる。

```
+1 : ℕ → ℕ
+1 m = suc m

-- eval +1 zero
-- return suc zero
```

Listing 3: Agda における関数定義

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 Code 4 のようになる。

```
_+_ : ℕ → ℕ → ℕ
zero + m = m
suc n + m = suc (n + m)
```

Listing 4: 自然数での加算の定義

パターンマッチでは全てのコンストラクタのパターンを含む必要がある。例えば、自然数 ℕ を受け取る関数では zero と suc の 2 つのパターンが存在する必要がある。なお、コンストラクタをいくつか指定した後に変数で受けることもでき、その変数では指定されたもの以外を受けることができる。例えば Code 5 の減算では初めのパターンで 2 つ目の引数が zero のすべてのパターンが入る。

```
_-_ : Nat → Nat → Nat
```

```
n - zero = n
zero - suc m = zero
suc n - suc m = n - m
```

Listing 5: 自然数の減算によるパターンマッチの例

Agda には λ 計算が存在している。λ 計算とは関数内で生成できる無名の関数であり、\arg1 arg2 → function または λarg1 arg2 → function のように書くことができる。Code 3 で例とした +1 をラムダ計算で書くと Code 6 のλ+1 ように書くことができる。この二つの関数は同一の動作をする。

```
+1 : ℕ → ℕ
+1 n = suc n -- not use lambda

λ+1 : ℕ → ℕ
λ+1 = (\n → suc n) -- use lambda
```

Listing 6: Agda におけるラムダ計算

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 Code 7 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。

```
f : Int → Int → Int
f a b c = (t a) + (t b) + (t c)
  where
    t x = x + x + x

f' : Int → Int → Int
f' a b c = (a + a + a) + (b + b + b) + (c + c + c)
```

Listing 7: Agda における where 句

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。Code 8 で書かれた、loop と stop は任意の自然数を受け取り、0 になるまでループして 0 を返す関数である。loop では ℕ の数を受け取り、 loop 自身を呼び出しながら 数を減らす関数 pred を呼んでいる。しかし、loop の記述では関数が停止すると言えないため、定義するには{-# TERMINATING #-}のタグが必要である。stop では自然数がパターンマッチで分けられ、zero のときは zero を返し、 suc n のときは suc を外した n で stop を実行するため停止する。

```
{-# TERMINATING #-}
loop : ℕ → ℕ
loop n = loop (pred n)

-- pred : ℕ → ℕ
-- pred zero = zero
-- pred (suc n) = n

stop : ℕ → ℕ
stop zero = zero
stop (suc n) = (stop n)
```

Listing 8: 停止しない関数 loop、停止する関数 stop

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. 定理証明支援器としての Agda

Agda での証明では関数の記述と同様の形で型部分に証明すべき論理式、 λ 項部分にそれを満たす証明を書くことで証明を行うことが可能である。証明の例として Code Code 9 を見る。ここでの +zero は右から zero を足しても ≡ の両辺は等しいことを証明している。これは、引数として受けている y が Nat なので、 zero の時と suc y の二つの場合を証明する必要がある。

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

Listing 9: 等式変形の例

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 Code 10 の 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。

```
cong : ∀ (f : A → B) {x y} → x ≡ y → f x ≡ f y
cong f refl = refl
```

Listing 10: cong

また、他にも λ 項部分で等式を変形する構文がいくつか存在している。ここでは rewrite と ≡-Reasoning の構文を説明するとともに、等式を変形する構文の例として加算の交換則について示す。

rewrite では 関数の = 前に rewrite 変形規則 の形で記述し、複数の規則を使う場合は rewrite 変形規則 1 | 変形規則 2 のように |を用いて記述する。Code 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。

```
+-comm : (x y : ℕ) → x + y ≡ y + x
+-comm zero y rewrite (+zero {y}) = refl
+-comm (suc x) y = let open ≡-Reasoning in
```

```
  begin
  suc (x + y) ≡⟨⟩
  suc (x + y) ≡⟨ cong suc (+-comm x y) ⟩
  suc (y + x) ≡⟨ sym (+-suc {y} {x}) ⟩
  y + suc x ■

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

Listing 11: 等式変形の例 3/3

Code 11 では suc (y + x) *equiv* y + (suc x) という等式に対して *equiv* の対称律 sym を使って左右の項を反転させ y + (suc x) *equiv* suc (y + x) の形にし、y + (suc x) が suc (y + x) に変形できることを +-suc を用いて示した。これにより等式の左右の項が等しくなったため +-comm が示せた。

Agda ではこのような形で等式を変形しながら証明を行う事ができる。

4. Continuation based C と Agda

本章では CbC に対応した Agda を記述する際の手法を説明する。

4.1 GearsAgda 形式で書く agda

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。code 12 が例となるコードである。

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

plus-com : {l : Level} {t : Set l} → Env → (next :
     Env → t) → (exit : Env → t) → t
plus-com env next exit with vary env
... | zero = exit (record { varx = varx env ; vary =
     vary env })
... | suc y = next (record { varx = suc (varx env) ;
     vary = y })

{-# TERMINATING #-}
plus-p : {l : Level} {t : Set l} → (env : Env) → (
     exit : Env → t) → t
plus-p env exit = plus-com env ( λ env → plus-p env
     exit ) exit

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

Listing 12: Agda での CodeGear の例

1 行目で Data Gear の定義を行っている。今回は 2 つの数値の足し算を行うコードを実装するため、varx と vary の二つの自然数を持つ。

7 行目の plus-com が受け取っている値を定義している。Env と next と exit を受け取っている。

next と next は Env → t となっているが、これは Env を受け取って不定の型 (t) を返すという意味である。これで 次の関数遷移先を取れるようにしている。

9 行目から 10 行目では入ってきた varx で場合分けを行っており、varx が zero ならそのまま vary を返し、次の遷移先へ、varx が zero 以外なら varx から 1 を引いて、vary に 1 を足して遷移する。

13 行目で x が zero 以外の値であった場合の遷移先を指定している。ここでは自身である plus-p をループするように指定した。CbC では再起処理を実装することはできないが、自己呼び出しを行うことはできるので、それに合ったように Agda でも実装を行なう。

17 行目が実際に値を入れる部分で、Exit が実行の終了になるようにしている。

前述した加算を行うコードと比較すると、不定の型 (t) により継続を行なっている部分が見える。これが Agda で表現された CodeGear となり、本論では Gears Agda と呼ぶ

4.2 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。今回はその Meta Gears を Agda による検証の為に用いる。

- 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。これを用いることで、仕様となる制約条件を記述することができる。
- 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。故に、Meta CodeGear は Agda で記述した CodeGear の検証そのものである

参考文献

[1] : The Agda wiki, http://wiki.portal.chalmers.se/agda/pmwiki.php. plus.33emminus.07emAccessed: 2018/12/17(Mon).
[2] : Agda1, https://sourceforge.net/projects/agda/. plus.33emminus.07emAccessed: 2020/2/9(Sun).
[3] : ATS-PL-SYS, http://www.ats-lang.org/. plus.33emminus.07emAccessed: 2020/2/9(Sun).
[4] : cbc-gcc - 並列信頼研 mercurial repository, http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_gcc/.

plus.33emminus.07emAccessed: 2020/2/9(Sun).
[5] : cbc-llvm - 並列信頼研 mercurial repository, http://www.cr.ie.u-ryukyu.ac.jp/hg/CbC/CbC_llvm/. plus.33emminus.07emAccessed: 2020/2/9(Sun).
[6] : Coq Source, https://github.com/coq/coq. plus.33emminus.07emAccessed: 2020/2/9(Sun).
[7] : Example - Hoare Logic, http://ocvs.cfv.jp/Agda/readmehoare.html. plus.33emminus.07emAccessed: 2019/1/16(Wed).
[8] : Hoare Logic in Agda2, https://github.com/IKEGAMIDaisuke/HoareLogic. plus.33emminus.07emAccessed: 2020/2/9(Sun).
[9] : loopSemInduct - 並列信頼研 mercurial repository, http://www.cr.ie.u-ryukyu.ac.jp/hg/Members/ryokka/HoareLogic/file/tip/whileTestGears.agda. plus.33emminus.07emAccessed: 2020/2/9(Sun).
[10] : Rust programming language, https://www.rust-lang.org/. plus.33emminus.07emAccessed: 2020/2/9(Sun).
[11] : Welcome! | The Coq Proof Assistant, https://coq.inria.fr/. plus.33emminus.07emAccessed: 2020/2/9(Sun).
[12] : Welcome to Agda's documentation! — Agda latest documentation, http://agda.readthedocs.io/en/latest/. plus.33emminus.07emAccessed: 2018/12/17(Mon).
[13] : whileTestPrim.agda - 並列信頼研 mercurial repository, http://www.cr.ie.u-ryukyu.ac.jp/hg/Members/ryokka/HoareLogic/file/tip/whileTestPrim.agda. plus.33emminus.07emAccessed: 2020/2/9(Sun).
[14] Hoare, C. A. R.: An Axiomatic Basis for Computer Programming, *Commun. ACM*, Vol. 12, No. 10, p. 576–580 (online), DOI: 10.1145/363235.363259 (1969).
[15] Kaito, T. and Shinji, K.: Implementing Continuation based language in LLVM and Clang, *LOLA 2015, Kyoto* (2015).
[16] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S.: seL4: Formal Verification of an Operating-system Kernel, *Commun. ACM*, Vol. 53, No. 6, pp. 107–115 (online), DOI: 10.1145/1743546.1743574 (2010).
[17] Moggi, E.: Notions of Computation and Monads, *Inf. Comput.*, Vol. 93, No. 1, pp. 55–92 (online), DOI: 10.1016/0890-5401(91)90052-4 (1991).
[18] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E. and Wang, X.: Hyperkernel: Push-Button Verification of an OS Kernel, *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, New York, NY, USA, ACM, pp. 252–269 (online), DOI: 10.1145/3132747.3132748 (2017).
[19] Norell, U.: Dependently Typed Programming in Agda, *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, New York, NY, USA, ACM, pp. 1–2 (online), DOI: 10.1145/1481861.1481862 (2009).
[20] Stump, A.: *Verified Functional Programming in Agda*, Association for Computing Machinery and Morgan & Claypool, New York, NY, USA (2016).
[21] 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).
[22] 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).
[23] 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，修士論文，琉球大学大学院理工学研究科 情報工学専攻 (2019).
[24] 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，Vol. 2018, pp. 197–206 (2018).
[25] 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).
[26] 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，Vol. 2012, pp. 69–78 (2012).
[27] 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).
[28] 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).