Agda と継続を用いたプログラムの検証
Verification of program using Agda and continuation

145750B 氏名 外間政尊 指導教員：河野 真治

Abstract

Program has high reliability. it is important. we are proposing to write program in units of CodeGear, DataGear for increase the reliability. we are deceloping Continuation based C (CbC) that can use units CodeGear and DataGear. In CbC, the handling of data in existing implementations is complicated. for that purpose, we can provide a interface mechanisms which are packages of CodeGears and DataGears. we made these units and interface available for Agda. Also converted Stack and Tree wrote in CbC to Agda. In this papaer, we tried several proofs on them.

1 ソフトウェアの信頼性の保証

ソフトウェアの信頼性を保証することは重要である。現在ソフトウェアの信頼性を保証する方法として代表的なものはモデル検査と、定理証明が存在している。当研究室では検証しやすいプログラムの単位として、CodeGear と DataGear 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。また、 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 (以下 CbC) を開発している。

CbC では通常の計算とメタ計算を分けて記述している。しかし、CodeGear で DataGear を扱う際、 Context と呼ばれる CodeGear、 DataGear のリスト等を持っている Meta DataGear を経由する。通常の CodeGear から Meta DataGear である Context を直接扱えると、ユーザーがメタ計算をノーマルレベルで自由に記述できてしまい、信頼性を損なう。そのため、 CbC では Context を通して、次の CodeGear に接続する MetaCodeGear である stub CodeGear を定義している。

CbC で実装していくにつれて特殊な stub CodeGear の記述が複雑になった。そこで既存の実装をモジュールとして扱うために Interface という仕組みを導入した。Interface では、DataGear に対しての操作 (API) を CodeGear とその CodeGear で扱われている DataGear の集合を抽象的に表現した Meta DataGear として定義されている。

本研究では CbC で使われている Interface を 、 Agda 上で定義、実装を行い、Interface を含めた Stack と Tree の部分的な証明を行なった。

2 Countinuation based C (CbC)

Continuation based C (CbC) とは、当研究室で開発されているプログラミング言語である。CbC は C 言語とほぼおなじ構文を持つが、 C の関数の代わりに CodeGear を用いて処理を記述する。CodeGear は関数定義の先頭に *__code* をつけて定義する。CodeGear は処理の単位でそれらの状態を goto で遷移して記述する。この goto による処理の遷移を継続と呼ぶ。DataGear は CodeGear が扱うデータの単位であり、処理に必要なデータが全て入っている。次の CodeGear に処理を移す際は、 goto の後に CodeGear 名と DataGear を指定する。CbC ではこの継続処理がメタ計算として定義されており、 CodeGear に変更なく検証等を行うことができる。例として CbC の簡単な例 (ソースコード 1) と、流れ 1 を示す。

ソースコード 1: CbC コードの例

```
__code cs0(int a, int b){
  goto cs1(a+b);
}

__code cs1(int c){
  goto cs2(c);
}
```

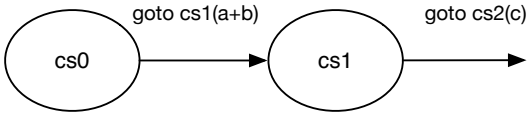

図 1: ソースコード 1 の流れ

ソースコード 1 では　cs0、 cs1 が CodeGear で a+b が cs0 の Output DataGear であり、cs1 の Input DataGear になる。流れ 1 は cs0 から cs1 へ継続した後、 cs0 には戻らずに次の継続に指定された CodeGear へ継続する。

3 CbC における Interface の定義

CbC で実装していくにつれ、stub CodeGear の記述が煩雑になった。そのため 既存の実装を モジュールとして扱うため Interface を導入した。Interface は DataGear に対して何らかの操作 (API) を行う CodeGear とその CodeGear

で使われる DataGear の集合を抽象化した メタレベルの DataGear として定義した。例として Stack での Interface の実装 (ソースコード 2) を示す。

ソースコード 2: CbC での Stack-Interface の実装

```
Stack* createSingleLinkedStack(struct Context* context)
    {
  struct Stack* stack = new Stack();
  struct SingleLinkedStack* singleLinkedStack = new
    SingleLinkedStack();
  stack->stack = (union Data*)singleLinkedStack;
  singleLinkedStack->top = NULL;
  stack->push = C_pushSingleLinkedStack;
  stack->pop  = C_popSingleLinkedStack;
  /* 途中省略 */
  return stack;
}
```

元の実装の push では Stack を指定する必要があるが、Interface での実装は push 先の Stack が stackImpl として扱われている。この stackImpl は呼ばれた時の Stack と同じになる。これにより、 ユーザーは実行時に Stack を指定する必要がなくなる。このように Interface 記述をすることで CbC で通常記述する必要がある一定の部分を省略し呼び出しが容易になる。

4 Agda

Agda[1] とは定理証明支援器であり依存型を関数プログラミング言語である。依存型とは型も第一級オブジェクトとする型システムであり、型の型や型を引数に取る関数、値を取って型を返す関数などが記述することができる。CbC を Agda に変換する場合 DataGear はレコード型、 CodeGear は Agda での通常関数として定義できる。前項で示した CbC の簡単な例を Agda に変換する。(ソースコード 3)

ソースコード 3: Agda における CodeSegment の定義

```
cs2 : CodeSegment ds1 ds1
cs2 = cs id

cs1 : CodeSegment ds1 ds1
cs1 = cs (\d -> goto cs2 d)

cs0 : CodeSegment ds0 ds1
cs0 = cs (\d -> goto cs1 (record {c = (ds0.a d) + (ds0.
    b d)}))

main : ds1
main = goto cs0 (record {a = 100 ; b = 50})
```

Agda のコードで関数を定義するときは関数名、型を記述した後に関数本体を指定する。関数の型では → または -> を使い定義し、関数本体は関数名の後に = をつけて記述する。DataGear はレコード型で表記できるため Agda 上で DataGear を定義することが可能である。このように CbC のコードを Agda に変換し、証明を行う。

5 Agda における Interface の定義、実装

Agda でも CbC と同様に Interface の定義、実装した。例として Agda で実装した Stack-interface の一部をみる。Stack の定義はソースコード 4、実装は ソースコード 5 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。

ソースコード 4: Agda における Stack -Interface の定義の一部

```
record Stack {n m : Level } (a : Set n ) {t : Set m } (si :
    Set n ) : Set (m Level.^^e2^^8a^^94 n) where
  field
    stack : si
    stackMethods : StackMethods {n} {m} a {t} si
  pushStack :  a -> (Stack a si -> t) -> t
  pushStack d next = push (stackMethods ) (stack ) d (\
    s1 -> next (record {stack = s1 ; stackMethods =
    stackMethods } ))
```

ソースコード 5: Agda における Stack -Interface 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}
```

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

今回は Interface を通した Stack で push 、 pop などの操作が正しく行われるかの証明を行った。ここでの証明とは Stack 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が、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」という性質を証明した。

まず始めに不定状態の Stack をソースコード 6 で定義した。 stackInSomeState が不定状態の Stack である。ソースコード 6 の証明ではこの stackInSomeState に対して、push を 2 回行い、 pop2 をして取れたデータは push したデータと同じものになることの証明している。

ソースコード 6: 抽象的な Stack の定義と push− >push− >pop2 の証明

```
stackInSomeState : {l m : Level } {D : Set l} {t : Set m
    } (s : SingleLinkedStack D ) -> Stack {l} {m} D {t
    } ( SingleLinkedStack  D )
stackInSomeState s =  record { stack = s ; stackMethods
    = singleLinkedStackSpec }
```

```
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 ) ) ))
push->push->pop2 {l} {D} x y s = record { pi1 = refl
       ; pi2 = refl }
```

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いた。関数本体で返る値は $x \equiv x1$ と $y \equiv y1$ で両方共に成り立つ為、refl で推論が通る。これにより、抽象化した Stack に対して push を 2 回行い、 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。

7 まとめ

本研究では CodeGear、 DataGear を用いたプログラミング手法を用いて、Agda で Interface を用いたプログラムを実装、検証した。また、 CbC で記述した時には細かく分かっていなかった Interface の型が明確になった。今後の課題としては、Tree 側では証明が複雑化し、うまく証明できていないことと、Hoare Logic 用いての証明を行えるように、CodeGear、DataGear をベースにした Hoare Logic を Agda 上で定義し、実際に証明を行うことなどが挙げられる。

参考文献

[1] The agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`. Accessed: 2017/10/24(Tue).

[2] 比嘉 健太, 河野 真治. メタ計算を用いた Continuation based C の検証手法, 2016.

[3] 伊波 立樹, 東恩納 琢偉, 河野 真治. Code Gear 、Data Gear に基づく OS のプロトタイプ, 2016.

[4] 徳森 海斗, 河野 真治. LLVM Clang 上の Continuation based C コンパイラ の改良, 2015.

[5] Welcome to agda ' s documentation! — agda 2.6.0 documentation. `http://agda.readthedocs.io/en/latest/index.html`. Accessed: 2017/10/24(Tue).