Agda による Equational Reasoning の証明の自動生成に向けて

山本 充子　(指導教員：浅井 健一)

1　はじめに

プログラムが正しいことを確認する方法として テストケースを作ることが一般的である。しかし、多くの場合においてテストケースに穴が生じ、結果としてバグが起きる。

このように、テストケースを作りそのプログラムの正しさを確認するよりも、証明されたプログラムの方が 100 % 正しいと言える。

プログラムの証明では、主に数学的規則を用いて証明し、定理証明系支援言語を使うのが一般的である。今回の証明では、定理証明系支援言語として Agda を用いる。

Agda を用いて、証明を行ったとき、時として何千行にもわたるプログラムになる。さらに、その証明の中には自明で機械的に行いたい部分もある。その機械的に証明できる部分を自動化することができれば、証明にかかる時間が短縮されるのではないかと考えられる。

Agda にも自動証明の機能はあるが、不十分な点が多い。そこで、今回は Agda で提供されている Reflection API [3] の機能を用いて自動証明を行う。さらにこの、Reflection API の `macro` という機能を用いて自分の定義した言語での自動証明を行える。しかし、この Agda で提供されている API はまだ広く普及しておらず、使い方などはあまり紹介されていない。Reflection API については、のちの節にて説明を行う。

2　λ 計算

今回の自動証明では、λ 計算 [2] を用いた自動証明を行う。λ 計算とは、1930 年に Alonzo Church によって、確立された枠組みで、関数の概念を定式化した計算モデルである。OCaml や Haskell などの関数型言語は この λ 計算に基づく。

今回は、定理証明系支援言語 Agda を使って、プログラムで証明する。論理式 (命題) の証明 と プログラムの証明 には、深い関係性がある。つまり、ある命題 A を プログラミング言語 (λ 計算) の 型 とみなし、ある命題 A の証明は 型 A を持つプログラム (λ 式) とみなすことができ、これを Curry-Howard 同型と言う。

この性質を利用して、Agda では、

関数名 : 関数の型（命題）
関数名 = 関数の定義（証明）

という形で記述し、λ 計算 の 簡約関係についての証明を行う。

3　対象言語の定義

自動証明を行うに当たって、Agda で λ 計算をどのように定義したのかを 項の定義、評価規則、簡約規則 [4] を順番に示す。

項の定義

$$\begin{array}{rrcl} \text{型} & t & := & \mathsf{int} \mid t \to t \\ \text{値} & v & := & n \mid x \mid \lambda x.\, e \\ \text{項} & e & := & v \mid e_1 \mathbin{@} e_2 \end{array}$$

評価規則

$$\begin{array}{rrcl} \text{フレーム} & F & = & [\,] \mathbin{@} e \mid v \mathbin{@} [\,] \\ \text{評価文脈} & E & = & [\,] \mid F\,[\,E\,] \end{array}$$

簡約規則

$$\frac{e_1 \to e_1'}{e_1 \mathbin{@} e_2 \to e_1' \mathbin{@} e_2}\ (\mathsf{App}_1) \qquad \frac{e_2 \to e_2'}{v_1 \mathbin{@} e_2 \to v_1 \mathbin{@} e_2'}\ (\mathsf{App}_2)$$

$$\frac{e[v/y] = e'}{\lambda x.\, e \to v}\ (\mathsf{RBeta})$$

これらの規則を用いてみると、

$$\begin{array}{ll} & ((\lambda x.\, x) \mathbin{@} (\lambda y.\, y)) \mathbin{@} ((\lambda x.\, x) \mathbin{@} 3) \\ \longrightarrow & \langle\ \mathsf{App}_1 \text{ を適用} \ \rangle \\ & (\lambda y.\, y) \mathbin{@} ((\lambda x.\, x) \mathbin{@} 3) \\ \longrightarrow & \langle\ \mathsf{App}_2 \text{ を適用} \ \rangle \\ & (\lambda y.\, y) \mathbin{@} 3 \\ \longrightarrow & \langle\ \mathsf{RBeta} \text{ を適用} \ \rangle \\ & 3 \end{array}$$

のようになる。

4　Equational Reasoning について

Equational Reasoning では、3 節で示した例のようにかく。つまり、$((\lambda x.\, x) \mathbin{@} (\lambda y.\, y)) \mathbin{@} ((\lambda x.\, x) \mathbin{@} 3)$ が 3 に 複数 step で簡約されることを、数学の方程式のように one-step ごとに簡約した結果をかくような形式である。さらに ⟨ ⟩ の中に one-step で簡約される際に適用する規則を入れる。このように表すことで、コンパクトに そして どのような手順で証明していったのかをわかりやすく読むことができる。上の例の証明を Agda で表すと、以下のようになる。

```
test4 : {var : typ → Set} →
  Reduce* {var}
  (App (App termx' termy) (App termx term3))
  term3
test4 =
  begin
    App
      (App (Val (Abst (λ x → Val (Var x))))
           (Val (Abst (λ y → Val (Var y)))))
      (App (Val (Abst (λ x → Val (Var x))))
           (Val (Num 3)))
  ⟶⟨ RFrame (App₁ (App termx term3))
            (RBeta (sVal sVar=)) ⟩
    App (Val (Abst (λ y → Val (Var y))))
        (App (Val (Abst (λ x → Val (Var x))))
             (Val (Num 3)))
  ⟶⟨ RFrame (App₂ (Abst (λ y → Val (Var y))))
            (RBeta (sVal sVar=)) ⟩
```

```
  App (Val (Abst (λ y → Val (Var y))))
      (Val (Num 3))
⟶⟨ RBeta (sVal sVar=) ⟩
  Val (Num 3)
■
```

5 Reflection API を使った自動証明

Agda では、reflection という機能を用いてメタプログラミングを行うことができる。reflection を用いることで、プログラムの本当のコード (Agda の内部表現) をプログラムで操作可能な構文木に変え、プログラムのコードを自動生成することが可能になる。

実際に、Reflection API を用いて自動証明をするには、macro と呼ばれるものを定義する必要がある。macro については、次節で説明を行う。

6 macro の定義

先ほどの、λ 計算の定義から自動証明を行うための macro の定義について説明する。

一般的に Agda で証明するときは、

```
関数名 : 関数の型
関数名 = { }
```

と書く。また、{ } は Hole と呼ばれ、Agda のコマンドを用いて Hole 内の Goal の型を見ることができる。その Goal の型を みて、Hole の中に当てはまる規則を書いていく。

macro は、reflection で用意された関数を用いて Goal の型を構文木として受け取り、Goal の型に当てはまる証明を返す機能を持つ。また、 Hole の中で何を記述するかは 2 節の定義から機械的にかける。そこで、macro の機能を使って、どう機械的に記述ができるのか示して、Hole に入れるべき証明を埋めるように指示をする。

これらをまとめたコードを示すが、ページの関係上、大体のアルゴリズムで示す。

```
counter-reduce : (n : ℕ) →
   (hole : Term) → TC ⊤
counter-reduce zero hole =
   typeError (strErr "time out" :: [])
counter-reduce (suc n) hole = inferType hole >>=
  λ { (def (quote Reduce*) args) →
       _⟶⟨_⟩_ をunify
          one-step で簡約されるなら、
             ⟨ ⟩ の中を再帰してunify
          one-step で簡約されないなら、
             ⟨ ⟩ の後のhole を再び_⟶⟨_⟩_ の形に
                  して、
             ⟨ ⟩ を再帰してunify した後、
             ⟨ ⟩ の後を再帰してunify
    ; (def (quote Reduce)
          (_ :: _ :: arg _ a :: _ : []))
       a がApp (e₁, e₂) で
             e₁ がApp ならApp₁ をunify
             e₁ がVal で
                    e₂ がApp ならApp₂ をunify
                    e₂ がVal ならRBeta をunify
     ; (def (quote Subst)
          (_ :: _ :: _ :: arg _ a :: _ :: _ :: []))
       a が(λ x → Val _) ならsVal をunify
       a が(λ x → App _) ならsApp をunify
     ; (def (quote SubstVal)
          (_ :: _ :: _ :: arg _ a :: _ :: _ :: []))
       a がVar ならsVar= をunify
       a が(λ x → Var x) ならsVar= をunify
       a が(λ x → Var y) ならsVar≠ をunify
       a が(λ x → Abst _) ならsFun をunify
       }
  ; t →
       typeError (strErr "not a reduction" :: [])
  }
```

macro の関数の概要を簡単に説明をする。

まず、手で証明するときと同様に、関数 inferType hole を用いて Hole の Goal の型を見る。今回、証明する λ 計算の Hole の Goal の型は、Reduce* と Reduce と Subst と SubstVal の 4 つに場合分けされるので、それぞれの場合についてパターンマッチを行う。

はじめに Reduce* A B の形をしていた場合は、Equational Reasoning の形になるように、_⟶⟨_⟩_ を埋めるように unify という関数を用いて指示する。one-step 簡約されない ならば、⟨ ⟩ の後の Hole が _■ を unify できるまで、counter-reduce を再帰して one-step 簡約の式をつなげる。

次に、Goal の型 が Reduce A B の場合である。これは、先ほどの Reduce* のときに生成された ⟨ ⟩ の Goal の型である。ここでは、A から B に one-step 簡約されるとき Hole に何を埋めるのかを指示する。簡約される前の A が 関数適用 (e_1 @ e_2) の形 で、e_1 が 関数適用 なら App_1 を unify する。さらに、e_1 が値 (数字または変数または λ 抽象) の形で、e_2 が 関数適用の型を持っているならば、App_2 を unify する。また、e_1 と e_2 がともに 値の場合は、RBeta を unify する。

このように、体系的にまとめることができ、比較的自動証明の方法を分かりやすく書くことができる。

7 自動証明の結果

6 節で定義した macro を用いて実行すると、4 節で示したような、one-step ごとの 証明が導出される。λ 計算を用いた、足し算の自動証明 ((1+2) + (3+4) → 10) などもできるようになっている。

8 まとめ

今回は、λ 計算の自動証明を行なった。現段階では、簡単なプログラムの例での自動証明しかしておらず、証明の行数もさほど多くない。今後の課題として、項の定義に shift/reset [1] を加え、CPS 変換の自動証明を考えている。

また、この自動証明の関数の定義を記述していると、機械的に表せる部分が多く、いずれ macro の関数のコード自体を自動生成できる macro が作ることができるのではないかと考えている。

参考文献

[1] O. Danvy and A. Filinski. Abstracting Control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160, 1990.

[2] Rob Nederpelt and Herman Geuvers. *Type Theory and Formal Proof: An Introduction*. Cambridge University Press, 2014.

[3] Agda Development Team. Reflection - agda 2.6.0.1 documentation. 2005–2019.

[4] 石尾千晶, 山田麗, 浅井健一. Agda による phoas を用いた CPS 変換の正当性の証明. *PPL*, Vol. 17, No. 1, pp. 3–4, 2018.