型理論での形式的証明記述の技法について

Some tips of theorem proving in type theory

木下　佳樹　　高村　博紀

Yoshiki Kinoshita　Hiroki Takamura

産業技術総合研究所　システム検証研究センター

Research Center for Verification and Semantics (CVS),
National Institute for Advanced Industrial Science and Technology (AIST)

yoshiki@ni.aist.go.jp　takamura-h@aist.go.jp

Martin-Löf 型理論での形式的証明の書き下ろしに用いられる技法, とくに等号の取り扱いかたなどを, 定理証明支援系 Agda の記法を用いて紹介する. 本稿の内容は, 定理証明支援系の利用者の間では folklore として伝わっているものが殆どであるが, まとまった形で出版されていなかったと思われる.

1　Agda の基本概念と記法

Agda[2] は Martin-Löf 型理論 [4] を基にした定理証明支援系である. 本稿では, 支援系自身だけでなく, Agda における型や項の記述言語も Agda の名で呼ぶこととする.

Agda の記述言語は, 一般化された帰納的定義, 依存積（函数型）, 依存和（シグネチャとストラクチャ）の三つの考えを基本としている.

帰納的定義は函数型プログラミング言語でもおなじみのものである. 例えば, 二つの真偽値, 真 $TT0$ と偽 $FF0$ を $\wedge$ で組み合わせて出来るような論理式の全体は, Agda によって

```
data form0:: Set
  = TT0
  | FF0
  | and0 (P,Q:: form0)
```

と定義される. 原子論理式の集合 X によってパラメータ化することもできる.

```
data form (X:: Set):: Set
  = TTT
  | FFF
  | at (p:: X)
  | and (P,Q:: form X)
```

ここで, 型 form はパラメータ X を持つが, X の値が構成子によって変わることがない. TT0 も at も and も, それらが返す値の型は form X である. しかし, 構成子によってパラメータを変えたい場合もある. その例として, 与えられた論理式の形式的証明を帰納的に定義することを考える. R を form X 型の値とするとき, R を帰結とする形式的証明の全体 proof X R を考えよう. Agda では, 公理や推論規則を proof X の構成子として,　例えば次のような定義が得られる.

```
proof (X:: Set):: form X -> Set
  = idata triv:: _ TT
        | andI (P,Q:: form X)
               (p:: proof X P)
               (q:: proof X Q)
          :: _ (and P Q)
        | andEl (P,Q:: form X)
                (p:: proof X (and P Q))
          :: _ P
        | andEr (P,Q:: form X)
                (p:: proof X (and P Q))
          :: _ Q
```

推論規則によって帰結の形が異なるから, 構成子によってパラメータが変わっていく. 例えば, 推論規則 andI($\wedge$-導入) を考えると, これは二つの論理式 P, Q およびそれらの証明 p, q が与えられたときに, $P \wedge Q$ の証明を返すものである. そこで, andI という構成子を導入して, $P, Q \in$ form X, $p \in$ proof X P, $q \in$ proof X Q のとき andI P Q p $q \in$ proof X (and P Q) と定める.　同様に $\wedge$- 除去左規則に相当する andEl という構成子を導入し, $P, Q \in$ form X, $p \in$ proof X (and P Q) のとき

andEl $P\ Q\ p \in \text{proof}\ P$ と定める．andI と andEl の型はどちらも proof にパラメータを渡した形になっているが，パラメータの値が異なっている．ここで用いられている定義は通常の帰納的定義よりも一般的なものなので，Agda ではキーワードも `data` ではなく `idata` とすることになっている．

依存積は [4] では $(\Pi a \in A)B(a)$ のように記されているもので，Agda では

```
(a:: A) -> B(a)
```

のように記す．とくに $B(a)$ が a に依存しない，つまり式 $B(a)$ において a が自由に出現しない場合には，`A -> B` と書く．依存積型の典型的な値はいわゆる λ 式 $(\lambda a \in A)b(a)$ で，Agda では

```
\(a:: A) -> b
```

のように記す．

シグネチャは，Martin-Löf 型理論の依存和に代わるもので，函数型プログラミング言語に用いられるシグネチャの拡張である．各フィールドの型が，そのフィールドより前に現れるフィールドの型に依存する（つまり，そのような型の変数が現れる）ことができる．

シグネチャの定義を述べる代わりに，依存和のシグネチャを用いた定義を与えて，シグネチャの使い方を示そう．

```
Sum (X:: Set)(Y:: X -> Set) :: Set
  = sig { fst:: X; snd:: Y fst}
```

シグネチャ型の項はストラクチャである．たとえば，$X :: Set, Y :: X \to Set$ が与えられているとき，次のように定義される pair によって，依存和型の項を与えることができる．

```
pair
  (X:: Set) (Y:: X -> Set)
  (x:: X) (y:: Y x)
  :: Sum X Y
  = struct { fst = x;  snd = y }
```

2 等号について

与えられた集合（Set 型の対象）に対して，等号を与える系統的な方法がいくつかある．しかし，どれも，あらゆる場合に満足のいくものではない．例えば，与えられた集合 X の上の Leibniz equality$\equiv$ は，「X 上の任意の述語 $P(x)$ に対して $P(a)$ が $P(b)$ を導く（含意する）とき，そのときに限り $a \equiv$b」と定義されるもので，Agda では次のように定義することができる．

```
LeibnizEq (|X:: Set) (x,y:: X):: Type
  = (P:: X -> Set) -> P x -> P y
```

Leibniz equality は確かに同値関係になる．しかし，Agda が準拠する Martin-Löf 型理論のように，「型全体の型」がなく，集合の全体の型 Set, Set を含む型全体がなす型 Type(#1 とも書く), Type を含む型全体の型 #2, ... と，型の宇宙 (universe) が累積的階層 (cumulative hierarchy) をなすような体系では，Leibniz equality は使いにくい．X が Set に属するとき，X 上の Leibniz equality は，Set より一段上の Type に属する．一般に $\#n$ に属する型 Y の上の Leibniz equality は，$\#(n+1)$ に属する．このため，複雑な型を構成すると，その型が属する宇宙はどんどん大きくなってしまうことになるからである．

もう一つの系統的な等号の与え方は，パラメータ付の帰納的定義を用いて，集合 X の上の等号 Id が成り立つことの証明を，Id x x にのみ与える，というものである．その証明は x をパラメータとする．Agda では，一般化された帰納的定義を用いて，Id を次のように定義することができる．

```
Id (|X::Set):: (a,b:: X) -> Set
    = idata ref (x::X) :: _ x x
```

Id では，等号の階層が上がってしまう，という問題は生じないが，今度は，外延性の問題が生じる．Id は，外延的ではない．つまり

```
(Y:: Set) -> (f,g:: X -> Y) ->
  ((x:: X) -> Id X (f x) (g x)) ->
                  Id (X -> Y) f g
```

が必ずしも成り立たない．外延性を持たない等号を用いた証明において，不自由が強いられることは，よく知られているとおりである．

系統的な等号の与えかたで，すべての点で満足できるものはない．そこで，等号を固定して考えることをやめ，その場その場で満足のいくような等号を用いることにし，いつも集合とその上の同値関係の対を考える，というアプローチがとられる [1]．このような対は Setoid とよばれ，Agda では次のように定義される．

```
Binrel (X:: Set):: Type = X -> X -> Set
```

```
Setoid :: Type
  = sig
    { Elem :: Set
    ; Equal :: Binrel Elem
    ; ref :: (x::Elem) -> Equal x x
    ; sym :: (x1::Elem) |->
             (x2::Elem) |->
             Equal x1 x2 ->
             Equal x2 x1
    ; tran ::
        (x1::Elem) |-> (x2::Elem) |->
        (x3::Elem) |->
        Equal x1 x2 -> Equal x2 x3 ->
        Equal x1 x3
    }
El (X::Setoid):: Set = X.Elem
Eq (|X::Setoid):: El X -> El X -> Set
  = X.Equal
```

集合の間の写像に当たるのが, Setoid **の間の** Fun **である**.

```
Fun (X::Setoid)(Y::Setoid) :: Set
  = sig { app:: El X -> El Y
        ; rsp:: (x1,x2:: El X) |->
                Eq |X x1 x2 ->
                Eq |Y (app x1) (app x2)
        }
```

Fun X Y **の要素は**, Setoid X (**正確には** El X) **から** Y **への写像で**, **付随する同値関係を保つようなものである**. Setoid **の上の単項演算や二項演算なども**, **次のように定義することができる**.

```
Zerop (X:: Setoid):: Set = El X
Unop (X:: Setoid):: Set = Fun X X
Binop (X:: Setoid):: Set
  = sig
    { app:: El X -> El X -> El X
    ; rsp:: (x1,x2,y1,y2:: El X) |->
            Eq|X x1 x2 -> Eq|X y1 y2 ->
            Eq|X (app x1 y1) (app x2 y2)
    }
```

代数系の定義を Setoid **を使ってやってみよう**. **たとえば単** (monoid) **は次のように定義される**.

```
MonoidAxiom (A:: Setoid) (id:: El A)
            (op:: Binop A)
  :: Set
  = sig
    { (&) = op.app
    ; (==) = Eq|A
    ; assoc
      :: (x1::(El A)) -> (x2::(El A)) ->
         (x3::(El A)) ->
           (x1 & x2 & x3)
           == (x1 & (x2 & x3))
    ; idl:: (x:: El A) -> (id & x) == x
    ; idr:: (x:: El A) -> (x & id) == x
    }
Monoid:: Type
  = sig { A:: Setoid
        ; op:: Binop A
        ; id:: El A
        ; ax:: MonoidAxiom A id op
        }
```

これだけの準備をすれば, **等式の証明を書くことができる**. **しかし**, **簡単な等式でも**, **それを形式的に書き下ろすと複雑なものになる**. **たとえば** Monoid **において**, a(b(cd))=((ab)c)d **が成り立つことの形式的証明を** Agda **で書いてみよう**. **定義のみから**, **補助的な手段を用いることを考えずに直接やると**, **例えば次のような証明** Dumb.proof **が得られる**.

```
package Dumb (M:: Monoid)
  where
  (&) = M.op.app
  proof:: (a,b,c,d:: El M.A) ->
          Eq |(M.A) (a & b & c & d)
                    (a & (b & (c & d)))
    =  \(a,b,c,d::M.A.Elem)->
       let
         (==) = Eq|(M.A)
         lem1:: (a & b & c)
                  == (a & (b & c))
           = M.ax.assoc a b c
         lem2:: (a & b & c & d)
                  == (a & (b & c) & d)
           = M.op.rsp lem1 (M.A.ref d)
         lem3:: (a & (b & c) & d)
                  == (a & (b & c & d))
           = M.ax.assoc a (b & c) d
```

```
      lem4:: (a & b & c & d)
               == (a & (b & c & d))
        = M.A.tran lem2 lem3
      lem5:: (b & c & d)
               == (b & (c & d))
        = M.ax.assoc b c d
      lem6:: (a & (b & c & d))
               == (a & (b & (c & d)))
        = M.op.rsp (M.A.ref a) lem5
      lem7:: (a & b & c & d)
               == (a & (b & (c & d)))
        = M.A.tran lem4 lem6
    in  lem7
```

この証明は読みにくい. **手で等式を証明するときには**,

$$u = v = x = y$$

のように, **等しい項を連ねて記し**, **推移律は暗黙のうちに使用するのだが**, Dumb.proof **では推移律が明示的に現われ**, **しかもそれらが他の推論と同じレベルで記されているのが**, **読みにくい理由の一つだと思われる**.

そこで, **例えば推移律の処理をライブラリで暗黙のうちに行なうことが考えられる**. **次のライブラリ** Chain **はその例である**.

```
package Chain
  (X:: Setoid)
  where
  (==):: Binrel (El X) = Eq|X
  chain (x,y:: El X):: Set
    = data ateq (p:: x == y)
         | ateq_inv (p:: y == x)
         | (=^) (z:: El X)
                (p:: x == z)
                (q:: chain z y)
         | (~^) (z:: El X)
                (p:: z == x)
                (q:: chain z y)
  ateq (|x,|y:: El X) (p:: x == y)
    :: chain x y
    = ateq@_ p
  ateq_inv (|x,|y:: El X) (p:: y == x)
    :: chain x y
    = ateq_inv@_ p
  (=^) (|x,|y,|z:: El X)
       (p:: x == z)
       (q:: chain z y)
    :: chain x y
    = (=^)@_ z p q
  (~^) (|x,|y,|z:: El X)
       (p:: z == x)
       (q:: chain z y)
    :: chain x y
    = (~^)@_ z p q
  ch_pr (|x,|y:: El X)
                (p:: chain x y)
    :: x == y
    = case p of
      (ateq p')-> p'
      (ateq_inv p')-> (X.sym p')
      ((=^) z p' q)->
        X.tran p' (ch_pr q)
      ((~^) z p' q)->
        X.tran (X.sym p') (ch_pr q)
```

このようなライブラリを手軽に書くことが出来るのは, Agda **が準拠している** Martin-Löf **型理論が証明と項を同一視し**, **必要に応じて証明をデータとして扱うことを容易にしているからである**.

Chain **を使った証明は**, **例えば次のようなものである**. lem1, lem2 **などが**, **通常の等式の連鎖になっているので**, **多少は読みやすい**. **それだけでなく**, **この連鎖を道しるべとするおかげで**, **証明の構築の途中で道に迷う**, **ということが少なくなる**.

```
package Wise (M:: Monoid)
  where
  open Chain M.A use chain, ch_pr,
       (=^), ateq
  (&) = M.op.app
  proof:: (a,b,c,d:: El M.A) ->
           Eq |(M.A) (a & b & c & d)
                     (a & (b & (c & d)))
    =  \(a,b,c,d::M.A.Elem)->
         let
           (==) = Eq|(M.A)
           lem1
             :: (a & b & c & d)
                == (a & (b & c) & d)
           = let
               lem1_1:: (a & b & c)
```

```
                == (a & (b & c))
             = M.ax.assoc a b c
         in M.op.rsp lem1_1 (M.A.ref d)
    lem2:: (a & (b & c) & d)
           == (a & (b & c & d))
      = M.ax.assoc a (b & c) d
    lem3:: (a & (b & c & d))
           == (a & (b & (c & d)))
      = let lem3_1:: (b & c & d)
                  == (b & (c & d))
            = M.ax.assoc b c d
        in M.op.rsp (M.A.ref a) lem3_1
  in
    ch_pr(lem1=^(lem2=^(ateq lem3)))
```

3 パラメータ付モジュール

よく知られているように, 依存積と依存和の二つの概念だけで, 強力なパラメータ機構を実現することができる. Agda **は, 既に** monoid **の定義でみたように**, package **というパラメータ付モジュール機構を提供しており, その意味は, 函数型とシグネチャに帰着させて説明することができる. この節では, パラメータ付モジュールを用いた定義の再利用の様子を紹介する**.

半環で, 加算が冪等律 $x + x = x$ **を満たす冪等半環を定義してみよう**. Kleene **代数** [3] **は冪等半環の重要な例である. 半環の定義には, 加算と乗算ふたつの** monoid **の構造がでてくるが, どちらにも, 先程定義した** MonoidAxiom **を適用することができる**.

```
IdempotentSemiringAxiom
  (A:: Setoid)
  (add, mul:: Binop A)
  (zero, one:: Zerop A)
  :: Set
  = sig { amon:: MonoidAxiom A zero add;
        ; mmon:: MonoidAxiom A one mul;
        ; (==) = Eq|A
        ; (+) = add.app
        ; (*) = mul.app
        ; acomm
          :: (x1, x2:: El A) ->
               (x1 + x2) == (x2 + x1)
        ; aidem
          :: (x:: El A) ->
               (x + x) == x
        ; cnstlft
          :: (x:: El A) ->
               zero * x == zero
        ; cnstrgt
          :: (x:: El A) ->
               x * zero == zero
        ; distrlft
          :: (x1,x2,x3:: El A) ->
               (x1 * (x2 + x3))
               == ((x1 * x2) + (x1 * x3))
        ; distrrgt
          :: (x1,x2,x3:: El A) ->
               ((x1 + x2) * x3)
               == ((x1 * x3) + (x2 * x3))
        }
IdempotentSemiring:: Type
  = sig { A:: Setoid
        ; add, mul:: Binop A
        ; zero, one:: Zerop A
        ; p:: IdempotentSemiringAxiom
              A add mul zero one
        }
```

冪等で可換かつ結合的な演算 $+$ **がある場合 (つまり** $+$ **に関する半束の公理が成り立っている場合) には, 半順序** $\leq$ **を**

$$a \leq b \iff a + b = b$$

によって定義することができるのはよく知られているとおりである. Kleene **代数に関する証明では, この半順序を用いると簡明になることも多い**.

さて, 等号に関するライブラリ Chain **を一般化して, 同様のライブラリを擬順序** (preorder) **に関して作ることができる**.

```
Rel (X:: Setoid):: Type
  = sig { rel:: El X -> El X -> Set
        ; cngr:: (x,y,u,v:: El X) |->
                 (p:: Eq|X x y) ->
                 (q:: Eq|X u v) ->
                 (r:: rel x u) ->
                  rel y v
        }
PreorderAx (X:: Setoid) (R:: Rel X):: Set
  = sig{ (<=) = R.rel
```

```
      ; ref:: (x:: El X) -> x <= x
      ; tran:: (x1,x2,x3:: El X)
            |-> x1 <= x2 -> x2 <= x3 ->
                                x1 <= x3
      }
package GChain
  (X:: Setoid)
  (R:: Rel X)
  (po:: PreorderAx X R)
  where
  (==):: Binrel (El X) = Eq|X
  (<=):: Binrel (El X) = R.rel
  data chain (x,y:: El X):: Set
    = at (p:: x <= y)
    | ateq (p:: x == y)
    | ateq_inv (p:: y == x)
    | (^)  (z:: El X) (p:: x <= z)
                      (q:: chain z y)
    | (=^) (z:: El X) (p:: x == z)
                      (q:: chain z y)
    | (~^) (z:: El X) (p:: z == x)
                      (q:: chain z y)
  chain_leqproof (|x,|y:: El X)
                 (p:: chain x y):: x <= y
    = case p of
      (at p')-> p'
      (ateq p')->
        R.cngr (X.ref x) p' (po.ref x)
      (ateq_inv p')->
        R.cngr
          (X.ref x) (X.sym p') (po.ref x)
      ((^) z p' q )->
        po.tran p' (chain_leqproof q)
      ((=^) z p' q)->
        po.tran
          (R.cngr (X.sym p') (X.ref z)
            (po.ref z)) (chain_leqproof q)
      ((~^) z p' q)->
        po.tran
          (R.cngr p' (X.ref z) (po.ref z))
          (chain_leqproof q)
```

同値関係は擬順序である. **したがって**, Chain **を**, **擬順序が等号と一致するような特別な場合として**, GChain **を用いて定義することもできる**.

ライブラリ GChain **を使うことによって形式的証明の記述が簡明になる様子を見るには**, **例えば乗算の単調性**

$$x \leq y, \qquad x' \leq y' \Longrightarrow xx' \leq yy'$$

の証明を GChain **を用いて行ってみるとよいであろう**.

謝辞

武山誠氏には, Agda **の詳細な利用法について御教示いただいた**. **また**, **本稿の内容においても**, **武山氏との日ごろの議論が資するところは大きい**. **記して感謝する**.

参考文献

[1] Bishop, E.: *Foundations of Constructive Analysis*, McGraw-Hill, 1967.

[2] Coquand, C.: Agda home page. `http://www.cs.chalmers.se/ catarina/agda/`.

[3] Kozen, D.: A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events, *Information and Computation*, Vol. 110(1994), pp. 366–390.

[4] Martin-Löf, P.: *An Intuitionistic Theory of Types*, Oxford University Press, 1998. (Reprinted version of an unpublished report from 1972).