National Institute of
Advanced Industrial Science
and Technology
AIST

AIST-PS-2008-014

Agda 言語について

木下佳樹

独立行政法人 産業技術総合研究所
システム検証研究センター

算譜科学研究速報
Programming Science
Technical Report

Research Center
for Verification and Semantics

独立行政法人 産業技術総合研究所 システム検証研究センター 発行
Published by Research Center for Verification and Semantics
National Institute of Advanced Industrial Science and Technology

Agda 言語について

Agda language

木下　佳樹

Yoshiki Kinoshita

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@m.aist.go.jp

構成的型理論に基づく Agda 言語を, 依存型を持つ函数型プログミング言語としての見方と, 論理を符号化するための言語としての見方の二つの見方から紹介する.

1　はじめに

Agda[1] は Chalmers 工科大学において 1990 年代後半から開発されてきた証明支援系である. Catarina Coquand, 武山誠, Marcin Benke らによって開発され, 2002 年に武山が産総研システム検証研究センター (AIST/CVS) に移籍してからは AIST/CVS も開発, 保守に参加し外部自動証明系連結のための plug-in 開発などに貢献した. 2006 年以後, Ulf Norell, Nils Anders Danielsson, 武山誠, Marcin Benke らによって Agda システムはスクラッチから新たに構築しなおされ, その機会にシステムが受理する言語に大幅な変更が加えられた. しばらくの間は, これ以前の言語が Agda1, 以後の言語が Agda2 と呼ばれていたが, 最近では Agda 言語といえば Agda2 を指す.

筆者らは [8] において Agda1 の概略を紹介したが, 上記のような Agda 言語の進化にともない, それは現状に合わなくなってしまっている. そこで今回, Agda 言語 (Agda2) の概略を最新の情報に基づいて改めて紹介したい.

2　Agda の基本概念と記法

Agda は Martin-Löf に始まる構成的型理論 (constructive type theory) [3, 2, 4] に基づく定理証明支援系である. 構成的型理論におけるいわゆる Curry-Howard 対応によって, Agda 言語では, 型は命題であるとみることもできるし, データ型であるとみることもできる. また, 型に属する項を命題の証明とみることもできるし, データ型の項とみることもできる, とくに函数型の項はそのままプログラムと見ることができる. さらに, 証明のあいだの変換 (同等性) と, 項の変換がうまく対応している.

データ型と項を表わすものとみなせば, Agda 言語は函数型プログラミング言語である. いっぽう, 命題と証明を表わすものとみなせば, Agda 言語は, ある形式理論における命題と証明を記述する言語である. 本稿ではまず, Agda 言語をプログラミング言語とする見方から紹介し, 次に命題とその証明を記述する言語としての Agda 言語を紹介することにする.

3　依存型付函数プログラミング言語として

Agda 言語は函数プログラミング言語とみることができる. 実際, Agda システムはコンパイラも備えている. 単純型付 λ 計算 (symply typed λ calculus) に基づく ML や Haskell などの最近の函数型プログラミング言語と違って, Agda は構成的型理論に基づくため, より複雑な型を扱うことができる. ML や Haskell は型を動く変数を備えているが, これらの変数を束縛する限量子を備えていないため, 型変数の出現はすべて自由な出現に限られる. これに対して, Agda 言語は, 型変数を束縛する限量子に相当するデータ型やレコード型を備えている. これらはいわゆる依存型とよばれるもので, パラメータ付の型を完全に一般な形で取り扱うことができる. 型も値と同じように計算の対象とすることができ, 型をやりとりする函数を書くこともできる.

Agda のプログラムはいくつかの宣言をならべてモジュールとしてまとめたものである. 宣言には函数宣言, 型の帰納的定義を可能にするデータ型宣言[1], レ

[1] ここでデータ型と呼ぶのは Agda のキーワード `data` で始ま

コード型宣言, 函数宣言, 公準の四種類がある. 本節ではこのうち, 前の三つとモジュールについて説明する. 公準は Agda による論理の符号化に関連するので次節で説明する.

3.1 モジュール

Agda のプログラムファイルは

```
module JSSST2008Prog where
```

のような行で始まるモジュールをあたえる。当面は、この一行をお呪いのようにファイルの先頭に付ける、と考えるのがよい。一つのファイルに複数のモジュールを置くことは推奨されない。モジュールを入れ子にすることもできるが本稿ではあつかわない。

他のファイルにあるモジュールをもってくるには

```
import JSSST2008Basic
```

などとする. これによりシステムは JSSST2008Basic.agda という名のファイルをさがし、そこのモジュールを持ってくる。この段階ではモジュール名 JSSST2008Basic が見えるようになるだけである。このモジュール内で宣言された名標 (identifier) が見えるようにするには、さらに

```
open JSSST2008Basic
```

とする必要がある。

Agda のモジュールは名標の見え方を制御するものであり、class システムのようなプログラムの実体の生成消滅を制御する機能はもっていない。他のモジュールの中で宣言された名標の実体を別の名前で参照することもできる。

モジュール機能と、後に説明するレコード型は関連が深い。実際、両者のうちの片方の考えをもちいて、もう一方を説明することができる。

3.2 型の帰納的定義

Agda には, 予め用意された型は Set しかない[2]. Set は型の全体がなす型ともいうべきもので, この要素 (つまり型) をデータ型宣言による帰納的定義で導入することができる.

型の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, 型の値を構成する構成子を並べることが基本である. たとえば true と false の二つの値からなる Bool 型を次のように定義することができる.

```
data Bool : Set where
  true  : Bool
  false : Bool
```

true と false は Bool 型の構成子である.

データ型宣言は自分自身を再帰的に参照するものであってもよい. 自然数の型 Nat は, Agda では例えば以下のように定義することができる.

```
data Nat : Set where
  zero : Nat
  succ : Nat -> Nat
```

型 Nat が導入され, 同時にその構成子として zero と succ が導入されている. zero は引数をとらない定数だが, succ は一引数の構成子である. したがって zero, succ zero, succ(succ zero) などは Nat 型の項で, それぞれ 0, 1, 2 をあらわす.

ここで, 字句と構文について若干触れておく. Agda では字下げ (indentation) や改行に意味がある. 上記で, 二行目と三行目が字下げされていることが必要で

```
data Nat : Set where
zero : Nat
succ : Nat -> Nat
```

は, 構文の間違いを含むものとして受理されない. また, 二行目と三行目の間の改行も必要で,

```
data Nat : Set where
  zero : Nat  succ : Nat -> Nat
```

も構文の間違いとなる. ただ, 行を改めるかわりにセミコロンで繋いで

```
data Nat : Set where zero : Nat ; succ : Nat -> Nat
```

あるいは

```
data Nat : Set where
  zero : Nat ; succ : Nat -> Nat
```

とすることもできる.

名標は, 英文字や数字だけからなる文字列と規定する言語が多いが, Agda では, unicode の文字をすべて名標に使うことができ, すぐ下で説明する mixfix 記

る特定の型宣言で導入する型, という意味で, データの型, という一般的な意味ではない.

[2] 詳しくいうと Set は Set0 であり, Set0 を要素とする Set1, Set1 を要素とする Set2 云々が用意されているが, これらについては後に述べる

法と合わせて, いろいろな形の演算子を使うことができる. その代わり, 名標の始めと終わりを空白文字によって明示しなければならない. たとえば Nat: Set と書くと Nat:と Set の二つの名標の列だということになる. Nat が Set 型の値だと書くには Nat と:の間に空白をいれて Nat : Set と書く必要がある.

Unicode の入力のためには, 例えば emacs の入力方法を TeX に指定するとよい. M-xset-input-method と打鍵すると Select input method (default japanese) などときいてくるから, これに対して TeX と入力する. 以後はその編集バッファで, TeX のコマンド名によっていろいろな記号の unicode を入力することができる. 例えば\prime とするとプライムの記号 (16 進で 0x539b2) を入力することができる.

succ は prefix の演算子であるが, Agda ではいわゆる mixfix 記法によるデータ型の構成子や函数記号などを導入することができる. そこで, zero のかわりに, (日本語漢字コードの!) 0 を使い, succ のかわりに postfix の 'を使って

```
data Nat : Set where
  0 : Nat
  _' : Nat -> Nat
```

とすることができる[3]. 三行目は 'が postfix の一引数オペレータで, Nat 型の値を引数として受け取って Nat 型の値を返すものであることを指定する. 以下では, これを Nat の定義として採用することにしよう.

データ型宣言によってパラメータ付の型を導入することもできる. たとえば型 A をパラメータとして, A 型の値を有限個ならべたものからなる型をつくる List を次のように定義することができる.

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A -> List A -> List A
```

ここで, [] は引数をとらない List A 型の構成子で, いわゆる空のリストをあらわす. ::は, A 型と List A 型の引数をひとつづつとって, 新しい List A 型の値を返す構成子で, いわゆる cons である. _::_と記すことによって, ::が infix の演算子であることを示す.

配列は, もっとこみいったパラメータ付の型, いわゆる依存型の定義の例である. 配列は長さが一定のリストだと見ることもできるので, List とよく似た定義をすることができる.

```
data Array (A : Set) : Nat -> Set where
  *    : Array A 0
  cons : (n : Nat) -> A -> Array A n ->
         Array A (n ')
```

Array A n は, 要素の型が A で, 長さ n の配列がつくる型である. 長さが 0 の配列というものは考えにくいが, *というたった一つの値を持つ型と考えることにしたのが二行目である. 三行目で三つの引数をもつ構成子 cons を導入する. cons n a x は, 長さ n の配列 x の端に a を付け足して得られる長さ n + 1 の配列である.

x がもつ型 Array A n は cons に渡される第一引数 n に依存している. そこで, cons の第一引数の型は単に Nat と書くのではなく, そこに渡される値をあとで参照できるように (n Nat):とする. これは, パラメータ付の型が制限された形でしか定義されない ML や Haskell などの函数型プログラミング言語にはなかったことである.

cons の第一引数 n をいちいち記すのは面倒である. 実際, n は第三引数の型 Array A n がわかっておれば, それから推測することができる筈である. そこで Agda には (n : Nat) の代わりに{n : Nat}と記して, cons の適用時に第一引数を省略することができるような機構が用意されている. つまり

```
data Array (A : Set) : Nat -> Set where
  * : Array A 0
  cons : {n : Nat} -> A -> Array A n ->
         Array A (n ')
```

と宣言すると, cons を適用するときに, はじめにくるはずの自然数の実引数を省略して cons a x と書くだけでよくなる. { }で囲って導入される仮引数は暗黙のうちに値が決まるので暗黙引数 (implicit argument) とよんでいる. これに対して通常の引数を明示引数 (explicit argument) という. 暗黙引数を明示的に指定することもでき, このときには cons {n} a x のように, 実引数を{ }で囲う.

さてこの場合, cons は二引数となるから, List の場合と同じように infix 記法を導入して

```
data Array (A : Set) : Nat -> Set where
  *   : Array A 0
  _^_ : {n : Nat} -> A -> Array A n ->
```

[3] ASCII の数字 0 は他の意味があるのでここでは使えない. また, ASCII の'(quote) も使えないので代わりにここでは日本語漢字コードの '(時間や角度の分をあらわす記号) を使っている.

```
Array A (n ′)
```

とすることができる. x に a を付け加えてえられる配列を a ^ x とあらわすことができる.

3.3 函数宣言

函数宣言は, 函数の名前を導入して, その型と値を指定する. 函数が持つ型は函数型と呼ばれる. 型 A の引数をうけとって, 型 C の値を返す函数の型を A -> C と書く.

例えば Bool 型に関して, 論理和や論理積, 反転などの基本的な函数を次のようにして定義することができる.

```
_\/_ : Bool -> Bool -> Bool
true  \/ _ = true
false \/ b = b

_/\_ : Bool -> Bool -> Bool
true /\ b = b
false /\ _ = false

neg : Bool -> Bool
neg true = false
neg false = true
```

函数と同様に定数も定義することができる[4]. 例えば以下のように 1 を定義することができる.

```
1 : Nat
1 = 0 ′
```

函数の再帰的な定義ももちろんできる. たとえばリストの長さを計算する函数 length を次のように定義することができる.

```
length : {A : Set} -> List A -> Nat
length [] = 0
length (_ :: as) = (length as) ′
```

length は二つの引数をとるが, 第一引数 A は前後関係から推測することにして, length を適用するときには一つの引数しか与えない, というのが{A : Set} の意味である. 実際, 第二引数の型 List A から A を推測することができる.

加算, 乗算, 階乗などを次のように定義することができる.

```
_+_ : Nat -> Nat -> Nat
n + 0 = n
n + (m ′) = (n + m) ′

_×_ : Nat -> Nat -> Nat
n × 0 = 0
n × (m ′) = (n × m) + n

_! : Nat -> Nat
0 ! = 1
(n ′) ! = (n !) × (n ′)
```

加算と乗算は infix 記号として, 階乗は postfix 記号として, それぞれ定義されていることに注意する.

さて, 既に前節に現われたように, Agda では型は一般にはパラメータを持っているから, 函数の結果の型が引数の値に依存する場合がある. 例えば, リストをその内容を変えずに配列に変換する LtoA を定義してみよう. 返す配列の大きさは, 受けとるリスト as の長さに等しくし, その型を Array A (length as) として, 次のようなものを定義することができる.

```
LtoA : {A : Set} ->
  (a : List A) -> Array A (length as)
LtoA [] = *
LtoA (a :: as) = a ^ (LtoA as)
```

ここで注意したいのは, 結果の型 Array A (length as) に引数 a が出現しており, したがって, 結果の型が実引数の値に依存して変化することである.

```
{x : A} -> (a : List A) ->
  Array A (length as)
```

のような型を依存型という. とくにこの場合のような函数型の依存型を依存積と呼ぶ.

依存型のもう一つの例として, 条件によって値を換える, いわゆる if then else の構成をあたえよう.

```
if_then_else_fi : {X : Set} ->
                  Bool -> X -> X -> X
if true  then A else _ fi  = A
if false then _ else B fi = B
```

if b then A else B fi は, b が true であれば A を, false であれば B の値をとる.

LtoA や if_then_else_fi は結果の型が実引数に依存する例だったが, 引数の型が, 他の実引数の値に

[4] 実は函数も函数型の定数に過ぎない.

依存する場合もある. 例えば, 配列の各要素に一様に函数を適用する ArrayMap を次のように定義することができる.

```
ArrayMap : (A : Set) -> (B : Set) ->
  (n : Nat) -> (A -> B) -> Array A n ->
  Array B n
ArrayMap A B 0 f * = *
ArrayMap A B (n ') f (a ^ as) =
  (f a) ^ (ArrayMap A B n f as)
```

しかし引数どうしの依存関係がある場合には, 片方を暗黙引数にしてもう片方から推測することのできる場合が多い. ArrayMap の場合も A, B, n を暗黙引数にして

```
ArrayMap : {A B : Set} -> {n : Nat} ->
  (A -> B) -> Array A n -> Array B n
ArrayMap f * = *
ArrayMap f (a ^ as) =
  (f a) ^ (ArrayMap f as)
```

とできる. ここで,

```
{A : Set} -> {B : Set} ->
```

と書く代わりに, これらを一まとめにして

```
{A B : Set} ->
```

と書いている点に注意されたい. 函数は Agda では

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

のように記す. 例えば, 「1 を加える」函数は

```
\(x : Nat) -> x + 1
```

と記される. map 函数を定義して, これに適用してみよう.

```
map : {A : Set} -> {B : Set} ->
      (A -> B) -> List A -> List B
map _ [] = []
map f (a :: as) = (f a) :: (map f as)

AddOneToElems : List Nat -> List Nat
AddOneToElems =
  map ( \ (x : Nat) -> x + 1 )
```

AddOneToElems は, 自然数のリストの各要素に 1 を加える.

いわゆる foldr 函数なども次のようにプログラムできる.

```
foldr : {A : Set} -> {B : Set} ->
        (A -> B -> B) -> B -> List A -> B
foldr _   b [] = b
foldr ope b (a :: as) =
  ope a (foldr ope b as)
Sum : List Nat -> Nat
Sum = foldr _+_ 0
```

Sum は, 自然数のリストの各要素の総和を計算する. なお, +のような infix 演算子を項として記すときには, ここでのように_+_などと, 引数の置き場所を指定する_を含めて記す.

3.4 レコード型

レコード型によって, いくつかの値をひとまとめにすることができる. レコード型の値は指定された型の値を一組にまとめたものである. 例えば, Bool 型の値と Nat 型の値の対を値とする BN 型を次のように定義することができる.

```
record BN : Set where
  field
    f1 : Bool
    f2 : Nat
```

f1 と f2 はフィールド名とよばれる. フィールド名によって, レコード型の対応する部分の値をとりだすことができる. 上記のレコード型の宣言は, 次の二つの函数の宣言を含む, と考えることができる.

```
BN.f1 : BN -> Bool
BN.f2 : BN -> Nat
```

これらの函数は射影 projection, あるいは選択子 selector と呼ばれる.

レコード型の式は次のようにしてつくられる: b が Bool 型の式で, n が Nat 型の式であれば,

```
record { f1 = b ; f2 = n }
```

は BN 型の式である.

レコード型は, Pascal や C, ML, Haskell などでもおなじみだが, Agda では, レコード型の各フィールドの型を, それより前に現われるフィールドの値に依存させることができる. 例えば次の VR 型は, Bool 型の配列とその長さをひとまとめにした値の集まりである.

```
record VR : Set where
  field
    len : Nat
    arr  : Array Bool len
```

vector フィールドの型は length に依存している. この場合, 選択子の型は次のようである.

```
VR.len  : VR -> Nat
VR.arr  : (r : VR) ->
          Array Bool (VR.len r)
```

帰納的定義の場合と同じように, レコード型にもパラメータを付けることができる. 例えば VR の Bool をパラメータにして

```
record PVR (X : Set) : Set where
  field
    len' : Nat
    arr' : Array X len'
```

を定義することができる. 選択子にもパラメータがつき, その型は次のようである.

```
PVR.len' : PVR -> {X : Set} -> Nat
PVR.arr' : (r : PVR) -> {X : Set} ->
           Array X (PVR.len' r)
```

レコード型へのパラメータ X は, 選択子の暗黙引数として扱われる.

4 論理の符号化

はじめに述べたように, Agda によって論理をはじめとする形式的体系を符号化 (encode) することができる. 一般に形式的体系を符号化するときに, 浅い符号化 (shallow embedding) と深い符号化 (deep embedding) の考えがある. 論理式の構造を Agda によって分解することができるのが深い符号化で, 分解できないのが浅い符号化, と大雑把に考えることができるが, 浅い符号化と深い符号化の両極端の中間に位置するような符号化も考えることができるので, これらを正確に定義することは難しい.

本節では以下で, まず一階述語論理の浅い符号化を示す.

4.1 浅い符号化—一階述語論理

直観主義に従って, 命題をその証明全体の集まりと同等のものと考えることにし, Set の要素を命題, さらにその要素がその命題の証明とみなす, という方針で, 一階述語論理を Agda によって符号化することができる. これが浅い符号化である.

偽 まず, 偽の命題⊥を導入しよう. 偽は証明がひとつもないような命題だから, 空集合によって表わす. 帰納的定義によって次のように表現することができる.

```
data ⊥ : Set where
```

こう書けば, ⊥の構成子は一つもなく, したがってその要素は一つもない.

どんな型 P をもってきても, ⊥型を引数として P 型を返す函数が一つだけ存在する. Agda ではそのような函数を次のようにして定義することができる.

```
⊥Elim : {P : Prop} -> ⊥ -> P
⊥Elim ()
```

ここで, () は存在しえない値にマッチする (したがってどんな値にも決してマッチない) パターンをあらわす. ⊥型の値は存在しないが, そのような, 存在しない値にマッチするパターンが () である.

連言 連言はレコード型を使って次のように表現できる.

```
record _∧_ (P Q : Set) : Set where
  field
    Elim1 : P
    Elim2 : Q
```

P と Q が命題であれば P ∧ Q も命題である. また P の証明と Q の証明を入力として P ∧ Q の証明を返す∧ Intro を次のように構成することができる.

```
∧Intro : {P Q : Set} -> P -> Q -> P ∧ Q
∧Intro a b =
  record { Elim1 = a ; Elim2 = b }
```

これは連言の導入規則に相当する. いっぽう, フィールド名から導かれる次の項は, 連言の除去規則に相当する.

```
∧.Elim1 : {P Q : Set} -> P ∧ Q -> P
∧.Elim2 : {P Q : Set} -> P ∧ Q -> Q
```

選言　選言は帰納的定義を使って次のように表現することができる.

```
data _∨_ (P Q : Set) : Set where
  ∨ Intro1 : P -> P ∨ Q
  ∨ Intro2 : Q -> P ∨ Q
```

P と Q が命題であれば P ∨ Q も命題である. 二つの構成子が二つの導入規則に相当する. 除去規則に相当する∨ Elim を次のように定義することができる.

```
∨ Elim : {P Q R : Set} ->
  P ∨ Q -> (P -> R) -> (Q -> R) -> R
∨ Elim  (∨ Intro1 a) prfP _    = prfP a
∨ Elim  (∨ Intro2 b) _    prfQ = prfQ b
```

含意　含意は函数型 P -> Q から帰納的定義を使って次のように表現できる.

```
data _⊃_ (P Q : Set) : Set where
  ⊃ Intro : (P -> Q) -> P ⊃ Q
```

構成子が導入規則に相当する. 除去規則に相当する⊃ Elim を次のように定義することができる.

```
⊃ Elim : {P Q : Set} -> P ⊃ Q -> P -> Q
⊃ Elim : (⊃ Intro f) a = f a
```

定義から明らかなように, P ⊃ Q 型の値の集まりから P -> Q 型の値の集まりへの一対一の上への写像が存在する. わざわざ構成子⊃を導入するのは他の論理記号と同じような取り扱いをするためである. ⊃ Elim は函数適用に他ならないので, 取り扱いの便利のために, infix の形も用意しておく.

```
_!_ : {P Q : Set} -> P ⊃ Q -> P -> Q
_!_ = ⊃ Elim
```

否定　含意をもちいて否定の論理記号と導入規則, 除去規則を次のように導入することができる.

```
¬ : Set -> Set
¬ P = P ⊃ ⊥
¬ Intro : (P : Set) -> (P -> ⊥) -> ¬ P
¬ Intro P pr = ⊃ Intro pr
¬ Elim : (P : Set) -> P -> ¬ P -> ⊥
¬ Elim P pr1 (⊃ Intro pr2) = pr2 pr1
```

以上で, 直観主義命題論理の符号化ができた. トートロジーの証明を二つばかり例示する.

```
p1 : (A B : Set) -> (A ∧ B) ⊃ (A ∨ B)
p1 A B = ⊃ Intro (\ (x : A ∧ B) ->
                  ∨    Intro1 (_∧_.    ∧
Elim1 x))
p2 : (A B : Set) -> (A ∧ B) ⊃ (A ∨ B)
p2 A B = ⊃ Intro (\ (x : A ∧ B) ->
                  ∨    Intro2 (_∧_.    ∧
Elim2 x))
```

全称限量子　全称限量子も含意と同様に (a P) -¿ Q a:を使って表現することができる.

```
data ∀ (P : Set)
       (Q : P -> Set) : Set where
  ∀ Intro : ((a : P) -> Q a) -> ∀ P Q
```

構成子が導入規則に相当する. 除去規則に相当する∀ Elim を次のように定義することができる.

```
∀ Elim : {P : Set} -> {Q : P -> Set} ->
         ∀ P Q -> (a : P) -> Q a
∀ Elim (∀ Intro f) a = f a
```

存在限量子　最後に存在限量子をレコード型によって次のように表現することができる.

```
record ∃ (P : Set)
         (Q : P -> Set) : Set where
  field
    evidence : P
    Elim     : Q evidence

∃ Intro : {P : Set} -> {Q : P -> Set} ->
         (a : P) -> Q a -> ∃ P Q
∃ Intro a prf =
  record { evidence = a ; Elim = prf }
```

これで一階述語論理の符号化ができた. 述語論理での証明の例を示す.

```
p3 : (A B : Set) -> (P : A -> B -> Set) ->
     (∃ A (\(x : A) ->
           ∀ B (\(y : B) -> P x y))) ⊃
     (∀ B (\(y : B) ->
           ∃ A (\(x : A) -> P x y)))
p3 A B P =
  ⊃ Intro (
  \(pr : ∃ A (\(x : A) ->
         ∀ B ( \(y : B) -> P x y))) ->
```

```
∀ Intro (\ (b : B) ->
  record {
    evi = ∃.evi pr ;
    prf = ∀ Elim (∃.prf pr) b }))
```

古典論理　二重否定除去 (Double Negation Elimination) あるいは排中律 (Law of Excluded Middle, LEM) を要請することを考えよう. ならない. Agda では公準 (postulate) によって型の値を導入することができる. これを用いて LEM の証明を一つ導入する.

```
postulate LEM : (P : Set) -> P ∨ (¬ P)
```

LEM を用いた, 二重否定除去の証明は例えば以下のとおり.

```
DNE : {A : Set} -> (¬ (¬ A)) ⊃ A
DNE {A} =
  ⊃ Intro (\(y : ¬ (¬ A)) ->
  ∨ Elim LEM
        (\(w : A) -> w)
        (\(z : ¬ A) ->
          (⊥ Elim (⊃ Elim y z))))
```

4.2　深い符号化—命題論理

浅い符号化では, 論理式は Set 型の値であったが, 与えられた一つの論理式の構造を, Agda プログラムの中で解析することができない. つまり, 与えられた論理式が原子論理式なのか, 何かの否定形なのか, 連言なのか, 選言なのか, 含意なのか, 全称限量子なのか, 存在限量子なのかを Agda のプログラムの中で判定して, それぞれに合った処理をする, ということができない. しかし, このような判定は, 論理式を対象にしたプログラミングでは非常に重要である. 論理式の構造を解析するのを可能にするのが「深い符号化」である. たとえば, 論理式全体がつくる型を帰納的に定義することによって, そのような解析を可能にする.

深い符号化は, より細かな処理が可能になる代わり, コーディングの量が増えてしまう. そこで, 以下では一階述語論理の一部分である命題論理をとりあげて, その深い符号化の例を示す.

論理式全体の型 Form を次のように定義する.

```
data Form : Set where
  Atom : Nat -> Form
  ⊥ : Form
  ¬ : Form -> Form
  _∨_ : Form -> Form -> Form
  _∧_ : Form -> Form -> Form
  _⊃_ : Form -> Form -> Form
```

可算個の命題変数 (propositional variables) があるとしており, n 番目の命題変数が `Atom` n である. ⊥は偽を表わす命題記号, ¬が否定, ∨, ∧, ⊃はそれぞれ選言, 連言, 含意をつくる論理記号である.

Form 型の値に対する等号を規定しておく. これは帰納的定義から機械的に得られるものだが, Agda システムは等号を用意しておらず, 利用者が定義しなければならない.

```
_==_ : Form -> Form -> Bool
(Atom m) == (Atom n) = eqN m n
(¬ P)    == (¬ P')   = P == P'
(P ∨ Q) == (P' ∨ Q') = (P == P') /\ (Q == Q')
(P ∧ Q) == (P' ∧ Q') = (P == P') /\ (Q == Q')
(P ⊃ Q) == (P' ⊃ Q') = (P == P') /\ (Q == Q')
_       == _         = false
```

こうすれば, たとえば, 真偽値計算のプログラムを書くことができる. ev は, 論理式と, 各命題変数への真偽値の付値φを受けとって, その付値のもとでの論理式の真偽値を返すものである. ここで, 引数である論理式の形に関する場合分けが行なわれていることに注意する. 浅い付値では, そのようなことができなかった.

```
ev : Form -> (Nat -> Bool) -> Bool
ev (Atom n) φ = φ n
ev ⊥        φ = false
ev (¬ P)    φ = neg (ev P φ)
ev (P ∨ Q) φ = ev P φ \/ ev Q φ
ev (P ∧ Q) φ = ev P φ /\ ev Q φ
ev (P ⊃ Q) φ = neg (ev P φ) \/ ev Q φ
```

浅い符号化のときのように, 自然推論による推論も符号化することができる. そのために, 各論理式の証明図全体がつくるデータ型 Proof を次のように定義する.

```
_＼_ : List Form -> Form -> List Form
Ps ＼ Q = del _==_ Ps Q

data Proof : List Form -> Form -> Set where
  Assume : (P : Form) -> Proof (P :: []) P
  ⊥ E : {fs : List Form} -> {P : Form} ->
        Proof fs ⊥ -> Proof fs P
  ¬ I : {fs : List Form} -> {P : Form} ->
        Proof fs ⊥ -> Proof (fs ＼ P) (¬ P)
  ¬ E : {fs gs : List Form} -> {P : Form} ->
        Proof fs P -> Proof gs (¬ P) ->
        Proof (fs $ gs) ⊥
  ∧ I : {fs gs : List Form} -> {P Q : Form} ->
        Proof fs P -> Proof gs Q ->
        Proof (fs $ gs) (P ∧ Q)
  ∧ E1 : {fs : List Form} -> {P Q : Form} ->
         Proof fs (P ∧ Q) -> Proof fs P
  ∧ E2 : {fs : List Form} -> {P Q : Form} ->
```

```
      Proof fs (P ∧ Q) -> Proof fs Q
  ∨ I1 : {fs : List Form} -> {P Q : Form} ->
      Proof fs P -> Proof fs (P ∨ Q)
  ∨ I2 : {fs : List Form} -> {P Q : Form} ->
      Proof fs Q -> Proof fs (P ∨ Q)
  ∨ E : {fs gs hs : List Form} ->
      {P Q R : Form} -> Proof fs (P ∨ Q) ->
      Proof gs R -> Proof hs R ->
      Proof ((fs $ (gs ＼ P)) $ (hs ＼ Q)) R
  ⊃ I : {fs : List Form} -> {P Q : Form} ->
      Proof fs Q -> Proof (fs ＼ P) (P ⊃ Q)
  ⊃ E : {fs gs : List Form} -> {P Q : Form} ->
      Proof fs P -> Proof gs (P ⊃ Q) ->
      Proof (fs $ gs) Q
  LEM : (P : Form) -> Proof [] (P ∨ (¬ P))
```

＼は Proof の定義のために必要な演算である. Ps ＼ Q は, 論理式のリスト Ps から, 論理式 Q を除去してえられるリストを返す演算子である. Ps の中に Q が二回以上現われていても, Ps ＼ Q の中には Q は現われない.

Proof Ps Q は, リスト Ps にある論理式を前提 (premise) とし, Q を帰結 (conclusion) とする証明図全体がつくるデータ型である. 証明規則毎に, Proof の構成子が一つずつあたえられている. たとえば Assume P は論理式 P を前提とし, それ自身を帰結とする証明図である.

個別の構成子に対する説明は, 自然演繹を知る読者には不要であろう. ただ, ここではプログラミングの簡単のために, 前提の解放 (discharge) の処理を単純にしている. 論理式 P を解放するときには, 前提のリストのなかのすべての P を解放する. 一般には, すべての P の出現を解放する必要はなく, その一部分 (0 個の出現でもよい) を解放するだけでよい, として十分である.

最後の構成子は, 排中律であり, LEM P は P ∨ ¬ P の証明図である. 直観主義論理を得たければ, この構成子を定義しなければよい. 浅い符号化では, LEM を postulate によって導入しなければならなかったが, ここでは, データ型宣言に一つの構成子を加えるだけでよい.

以上の設定のもとで, Atom 0 という, 特定の命題に関する二重否定除去の規則の証明を次のように書くことができる.

```
A : Form
A = Atom 0
DNE0 : Proof [] ((¬ (¬ A)) ⊃ A)
DNE0 = ⊃ I (∨ E (LEM A)
                (Assume A)
                (⊥ E (¬ E (Assume (¬ A))
                     (Assume (¬ (¬ A))))))
```

5 その他

5.1 宇宙 (universe))

Agda 言語には宇宙の階層

```
Set : Set1 : Set2 : ...
```

が用意されている. Set は Set1 の元であり, Set1 は Set2 の元である. この他の値は全く定義されていないが, 帰納的定義 (データ型宣言) によって導入することができる. これらは, 論理の符号化において用いられることが多く, 初歩のプログラミングでは殆ど必要が生じないと考えられる.

5.2 実現

Agda2 システムは Haskell によって実現されており, Linux と Windows で稼働している. また, emacs インターフェイスが用意されており, 利用者 Linux 版の darc および Windows 版のためのインストーラのリンクが Agda wiki ページ [1] にある. とくに Windows 版インストーラは, NTEmacs および必要な unicode フォントのインストールを含み, すぐに Agda の利用を開始することができる.

Agda2 のコンパイラ MAlonzo も用意されている. これは Agda2 プログラムを等価な Haskell プログラムに変換し, その後 Haskell コンパイラによってネイティブなコードに変換する. Agate[7] は Agda1 を対象にするものであったが, MAlonzo はその後 Agda2 を対象としてつくらてあものである.

5.3 文献

Agda2 の言語は現在のところ, まだ参照マニュアルが用意されていない. 最も整った紹介は, システム実現に中心的役割を果たした Ulf Norell の博士論文 [5] にある. Agda2 言語をプログラミング言語として紹介したチュートリアル [6] もある. Agda wiki ページ [1] において, Agda に関する最新情報を得ることができる.

参考文献

[1] Agda community: Agda wiki.
http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php?n=Main.HomePage.

[2] Martin-Löf, P.: On the meaning of the logical constants and the justification of the logical laws. short course given at the meeting Teoria della Dimostrazione e Filosofia della Logica, organized in

Siena, 6-9 April 1983, by the Scuola di Specializzazione in Logica Matematica of the Universita' degli Studi di Siena.

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

[4] Nordström, B., Petersson, K., and Smith, J. M.: *Programming in Martin-Löf's Type Theory, An Introduction*, Oxford University Press, 1990. Now out of print, but PDF and PostScript file available at `http://www.cs.chalmers.se/Cs/Research/Logic/book/`.

[5] Norell, U.: *Towards a practical programming language based on dependent type theory*, PhD Thesis, Chalmers University of Technology, 2007. PDF file available at `http://www.cs.chalmers.se/~ulfn/papers/thesis.pdf`.

[6] Norell, U.: Dependently Typed Programming in Agda, 2008. Unpublished draft; PDF file found at `http://www.cs.chalmers.se/~ulfn/darcs/AFP08/LectureNotes/AgdaIntro.pdf`.

[7] Ozaki, H., Takeyama, M., and Kinoshita, Y.: Agate—an Agda-to-Haskell compiler. to appear in Computer Software.

[8] 木下佳樹, 高村博紀: 型理論での形式的証明記述の技法について, **日本ソフトウェア科学会第** *22* **回大会予稿集**, 日本ソフトウェア科学会, 2005.

Agda 言語について
(算譜科学研究速報)
発行日：2008 年 9 月 10 日
編集・発行：独立行政法人 産業技術総合研究所 (システム検証研究センター)
同連絡先：〒560-0083 大阪府豊中市新千里西町 1-2-14 三井住友海上千里ビル 5F
TEL：06-4863-5025
e-mail：informatics-inquiry@m.aist.go.jp

Agda language (in Japanese)
(Programming Science Technical Report)
10 September 2008
(Research Center for Verification and Semantics (CVS))
National Institute of Advanced Industrial Science and Technology (AIST)
5F Mitsui Sumitomo Kaijo Senri Bldg., 1-2-14, Shinsenrinishi-machi, Toyonaka,
Osaka 560-0083 Japan
TEL：+81-6-4863-5025
e-mail：informatics-inquiry@m.aist.go.jp

AIST01-J00022-87