Agda での Programming 技術

河 野 真 治[†1]

Agda[1] は、Haskell[2] 上の Dependent type に基づく証明支援系である。Agda での証明は型付きλ式であり、証明するべき式も型付きλ式になっている。したがって、Agda での証明は、本質的にはλ式を構築していく作業になる。

この発表では Agda の入門と、Emacs 上で行われる Agda のプログラミング技術について圏論の例題を用いながら紹介する。Dependent type に関する論文は多いが、ここでは Agda の証明を構成するために使われる技術を紹介していく。

プログラミング言語と異なり、Agda の証明を書き上げても、それが動くことはない。その意味では、Agda の証明はプログラミングとは異なる。しかし、型宣言を行い、その型に沿ったλ式を書いていくという点ではプログラミングそのものである。

Agda の証明は、Curry Howard 対応[3] に基づいており、すべては、型を表す式と、その型を持つλ式の組で表される。Dependent type の特徴は省略可能な引数 (暗黙の引数) にあり、これは、通常のプログラミング言語にはない概念である。

証明を構成する作業は、型に沿って行われる。Agda には、未定部分を ? としして指定すると、その部分に相当する型をλ式として表示してくれる。これにより、証明の指針が得られる。

証明は型が十分に与えら得ていることが必要であり、未定の部分が含まれている場合には Agda は、その部分を黄色で表示する。これを解決するには、暗黙の引数を表示してやると良い。

また、Agda の特徴である式変形を明示する推論に対するいくつかの手法に関しても考察する。

Programming Technique in Agda

Shinji KONO [†1]

1. 証明支援系

証明支援系の歴史は古く、80 年代には ML で記述された HOL[4] が既に使われるようになってきた。ここで使う Agda[1] は、証明とλ式が対応するという Curry Howard 対応に基づく証明支援系であり、Haskell[2] で記述されている。この論文では、Agda での証明の実際をプログラミングの視点から考察する。

証明支援系は、従来、小さいプログラムが満たす性質を証明しようとしても記述が膨大になるという欠点があった。近年では、数万行程度のプログラムも証明の対象にすることができるようになってきてる。しかし、ここでは、実際のプログラムの性質の証明とは違った視点から証明支援系の必要性を調べてみたい。

Haskell などの関数型言語では、I/O や継続、計算量、並列計算などの計算の様々な様相を Monad の仕組み[5] を用いて実現している。Monad は圏論 (Category theory)[6] の概念であり、その性質の証明は若干煩雑である。Monad は随伴関手 (Adjoint pair) や、Kleisli 圏と対応しているが、それを理解するには一連の証明を理解する必要がある。

Monad の非数学的な説明はいくつかあるが、Monad を使ってプログラムを記述する必然性を理解しようとすれば証明を避けるのは好ましくない。そこで、証明支援系を使って、Monad を理解できれば、関数型プログラミングを学ぶものにとって都合が良いと思われる。特に、Haskell で記述された Agda は構文的に Haskell と似ていて、理解しやすい。

プログラムのバグを見つけるにはモデル検証[7] が使われるようになってきているが、モデル検証は式と値の対応に着目して、その可能な対応をすべて列挙することにより反例を見つける。これは、命題式に真偽を割り当てていく方法になっている。

Agda の論理は、証明を提示すことによって命題式が真であることを示すハイティング代数というものである。モデル検証とは本質的に異なる方法である。

証明支援系としては Coq も良く使われている。Agda と Coq は良く似ているが、Agda は Emacs を用いた式変形と穴埋め的な使い方をするのに対して、Coq

†1 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus.

は自動証明を重視したコマンドによる式変形を中心とした使い方するようになっている。証明支援系の比較などは、この論文[8] が参考になる。

ここでは Agda での証明の対象して圏論、特に、Monad を用いる。Monad を理解するために、Agda をどのように使えるか、そこで使える Agda でのいつくかの tips を示す。

2. 人による証明

Agda での証明の実際に入る前に、本に書かれているような証明について調べてみよう。

数学の本には証明が極めて簡潔に書かれている。Monad を理解するためには、圏 (Category)、関手 (Functor)、自然変換 (Natrual Transformation)、Monad が満たす関手と自然変換の性質を理解して、Haskell の Monad の join の結合法則

```
f * ( g * h ) = ( f * g ) * h
```

を証明することが目標だとしよう。ここでは Introduction to higher order categorical logic を例として用いる。

圏の定義は、

Definition 1.1. A concrete category is a collection of two kind of entities called object and morphisms. The former are sets which are endowed with some kind of structure, and the latte are mappings, tha is functions from one object to another, in some sense preserving that structure. Among the morphisms, there is atteched to each object A the identity mapping 1_A: A → A such aht 1_A(a) = a for all a ∈ A. More over, morphims f : A → B and g : b → c may be compoosed to producue a morphism gf: a → c such hat (gf)a() = g(f(a) for all a ∈ A.

となっている。ここでは、この細かい定義が重要なのではなくて、自然言語による定義と、あとで示す Agda での定義を対照させるために示している。 圏は、 object (対象) と morphism (射) を持っている。ここで重要なのは、(gf)a() = g(f(a) という結合の定義である。ここでは、object や morphism が何かははっきり書いてない。Agda では両方共、あるレベルの集合になる。 関手の定義は、

Definition 1.3. A cuntor F: A → B is first of all a morphism of graph, that is, it sends objects of A to object so B and arrows of A to arrows of B such that, if f: A → A' then F(f):F(A) → F(A'). Moreover functor preserves identities and compisition; thus

$F(1_A) = 1_F(A), F(gf) = F(g)F(f)$

であり、対象と射の二つの写像により関手が定義されることがわかる。

自然変換の定義は、

Definition 2.1. Given functors F,G: A → B, a natural transformatin t: F → G is a familty of arrows t(A): F(A) → G(A) in B, one arrow for each object A of A, suc that the following square commutes for all arrows f: A → in A. that is to say, such that

G(f)t(A) = t(B)F(f).

と書かれている。これが何かを理解することは、この定義だけからでは難しい。

さらに、Monad は関手 T と二つの自然変換の組であり以下のように定義されている。

tiny Definition 6.1 A triple(T, η, μ) on a category A consists of a functor T: A → A and natural transformation η: 1_A → T and μ:T^2 → T satifying the equation

μ oT η = 1_T = μ o η T, μ o μ T = μ oT μ

となっている。 ここで、関手 T と自然変換ηの 合成 T ηの定義は

```
(T η)(A) = T(η (A))
(η T)(A) = η (T(A))
```

である。これらの英語は、大学初年度の数学の知識があれば読んで理解することが期待されている。この本の記述は、既に圏論を知っている人向けに書かれており、特に簡潔に書かれている。以下に示す Agda の定義は、これらの 10 倍程度の量になる。

これらの定義から証明するべきなのは、join の結合性であり、join は以下のように記述される。

```
g * f =def μ (A)T(g)f
```

以上の定義に対して、

```
f * ( g * h ) = ( f * g ) * h
```

を示せば良い。これで問題は十分に定義されたはずである。しかし、ここから、この式を証明するまでの道のりはやさしくない。

本の証明は自然言語で書かれていて、簡潔な代わりに多くのものが省略されている。例えば、式?? に出てくる T と式?? に出てくる T は、前者は arrows から arrows の関数であり、後者は object から object への関数である。つまり、同じ T で二つの異なる関数を表している。確かに関手の定義には、二つの関数が記述されている。これは、文章による証明の記述では記号が polymorphic に使われることを示している。つまり、それが要求される型によって異なる関数が対応している。

2.1 関手とは何か、自然変換とは何か

関手は定義の通りのものだが、Monad の視点から見ると、データ型のコンストラクタに相当する。例えば、文字の集合があった時に、文字列やファイルを作るのは、文字の集合に相当する圏から、ファイルの集合に相当する圏への関手になる。この場合、関手は文字列の結合に対する分配則を満たす必要がある。

すると、自然変換はデータ型からデータ型への変換

を表す。例えば、SJIS のファイルから UTF8 へのファイルに変換するプログラム nkf は、自然変換だと考えられる。実際、 ファイルからファイルへの操作、例えば grep があった時に、SJIS で SJIS 対応の grep を使ってから、nkf で UTF8 の結果に変換するのと、先に nkf で UTF8 に変換してから UTF8 対応の grep を使うのと結果は同じでなければならない。これが可換性であり、自然変換であるための条件になる。ここでは自然変換の引数は UTF8 や SJIS の型に相当する。

```
grep word — nkf -w
nkf -w — grep word
```

ここで結果は同じであるべきだが、前者の grep は SJIS に対して動き、後者は UTF8 に対して動く。

```
G(f)t(A) = t(B)F(f).
```

G と F がファイルの型を表していて、この t が grep に相当する。引数 A,B は SJIS や UTF8 になる。射 f は、この場合は nkf である。

このような特定の分野に向いた例示は定義からは出てこない。

しかし、証明で重要なのは、関手の提供する関手の分配法則と、自然変換の提供する可観測であり、それ以外の余計な直観は必要ない。

2.2 Monad の join とは何か

Monad は一つの関手 T と二つの自然変換 (η, μ) を含む複雑な概念だが、T は Monad を表すデータ構造のコンストラクタだと考えて良い。これは、計算の様々な様相を表すメタデータと、通常の入力との組であることが多い。

η は、メタデータから通常のデータだけを取ってくる自然変換であり、Haskell Monad の return に相当する。μ は、結合されるべき二つの関数が生成する二つの T を一つの T にまとめる自然変換である。これは、Haskell Monad の join に相当する。

3. 手書きの証明

定義は与えられたので、証明するべき命題、

```
f * ( g * h ) = ( f * g ) * h
```

を定義に沿って展開すれば良い。これはかなり長い式になる。右側と左側から両方を展開する。そして、見比べる。これを関手の分配則と、自然変換が持つ可観測により変形して等しいことを示せば良い。

ここで、可換図が重要になる。可換図は自然変換の可換則の具体例を示すのに使える。つまり、自然変換の等式の変数を具体的に決めたものを与える。これを使って、計算することになる。

4. 証明の清書

手書きの証明が終わると、これを例えば LaTeX などで清書する。清書には一部可換図も含まれる。ここでは以下のような式の変形と可換図が得られる。

まず、左辺と右辺を定義に沿って展開する。

$$\begin{aligned} g * (f * h) &= g * (\mu(c)T(f)h) \\ &= \mu(d)(T(g))(\mu(c)T(f)h) \\ &= \mu(d)T(g)\mu(c)T(f)h \end{aligned}$$

$$\begin{aligned} (g * f) * h &= (\mu(d)T(g)f) * h \\ &= \mu(d)T(\mu(d)T(g)f)h \\ &= \mu(d)T(\mu(d))T^2(g)T(f)h \end{aligned}$$

これが等しいことは、μ の可換性からわかる。

$\mu(d)\mu(d)T = \mu(d)T\mu(d)$

$\mu(T(d))T^2(g) = T(g)\mu(c)$ naturality of μ.

$\mu(d)T\mu(d)T^2(g) = \mu(d)\mu(T(d))T^2(g) = \mu(d)T(g)\mu(c)$

可換図は以下のようになる。

$$\begin{array}{ccccc} T^2(d) & \xleftarrow{T\mu(d)} & T^2(T(d)) & \xleftarrow{T^2(g)} & T^2(c) \\ \downarrow{\scriptstyle \mu(d)} & & \downarrow{\scriptstyle \mu(T(d))} & & \downarrow{\scriptstyle \mu(c)} \\ T(d) & \xleftarrow{\mu(d)} & T(T(d)) & \xleftarrow{T(g)} & T(c) \end{array}$$

5. 証明を理解するということ

ここでは証明は式の一連の変形であり操作に過ぎない。この操作を理解することと、関手や自然変換が何かを理解することは別物である。

圏や関手の公理を満たす例を考えることは、その数学構造のモデルを考えることに相当する。モデルとは、ここでは、公理をすべて正しくするような変数の値の組である。

モデルとして圏や関手を理解することと、証明を理解することは違うことだが、モデルがあれば、その理論は整合的であることが言える。また、証明があれば、その式は正しい。

Monad の定義から自明に join の結合法則を導くようなモデルを考えることは難しいが、証明自体は式の変形でわかりやすい。しかし、量も多く煩雑な操作となる。ここで、証明支援系が役に立つと思われる。

6. Agda

ここでは Agda をざっと概観する。例えば、[1] の論文がわかりやすい。Agda は Haskell と同じような構文を持つλ式の集合である。例えば、Agda の List は以下のようになる。ここでの Agda の例題は[9] で見ることができる。

```
infixr 40 _::_
data List {a} (A : Set a) : Set a where
  [] : List A
  _::_ : A → List A → List A
```

これに対して append のプログラムは以下のようになる。

```
infixl 30 _++_
_++_ : ∀ {a} {A : Set a} → List A → List A → List A
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

このプログラムは、Haskell の定義とほぼ同じである。Haskell と違い型定義を省略することはできない。また、List の要素の型 A が Set として明示されている。最初の data List が List のデータ型を定義している。append は ++ で定義される。

つまり、Agda では、一つの関数を定義するのに、型と関数の両方を定義する必要がある。Agda の型は、また、Agda の式であり、それ自体の型の整合性が要求される。 一つの名前を定義するのに、: で型を定義して、= で、その値を定義する。

Haskell の型と異なり、Agda では、型と式に同じものを書ける。それだけ強力な型システム j を持っている。型の記述にも型の整合があり、値は、その記述された型に整合する必要がある。つまり、Agda での型の整合は一つの式の定義に対して二度行われることになる。

data 型の filed の数だけパータンマッチングを作ることができ、この場合は、[] と _::_ の二つの場合分けになっている。:: は中置き演算子であり、

```
_::_(a,b)
```

と

```
(a :: b)
```

は同じものを表している。

ここでは型には (変数名 : 型名) による型の要素の定義と、(入力型→出力型) で表される関数の二種類がある。

a は、暗黙の引数であり、呼び出す時に省略することができる。ここでは、集合のレベルを暗黙に与えている。∀ は飾りであるが任意の a に付いて成り立つことを意味している。

Agda では集合のパラドックスを避けるために集合の集合は一つの上のレベルの集合になる。Set a はレベル a の集合になる。Set (suc a) は、Set a の一つ上のレベルの集合を表す。ここでは、任意のレベルの集合に対する List を定義している。

infixl, infixr が中置き演算子であり、_ が引数の位置を表す。a ++ b の演算子は _++_ であり、_++_ a b と同じものを表している。

6.1 data 型を使った等号

Agda には、さまざまな等号や関係が定義されるが、同じ物を表すのには data 型が使われる。

```
infixr 20 _==_
data _==_ {A : Set } : List A → List A → Set where
  reflection : {x : List A} → x == x
```

同じ物を二つ並べて作られるデータ構造が reflection を提供する。これは等号の反射率 a= a に相当する。data の filed の reflection は、等号のコンストラクタだと考えることができる。つまり、a を与えると、a == a というデータ構造を返す。これは、a == a の証明に対応する λ 式でもあって、Curry Haward 対応に相当する。

これは、実際には、Relation .Binary .PropositionalEquality で _≡_ として定義されているので、自分で定義する必要はない。

data 型の他に record 型あり、この append の例題では使わないが、排他的和を表すデータ構造を提供し、圏や関手を定義することができる。

7. Agda での証明方法

Agda は Emacs と連携して使用されることが想定されている。Emacs が backend である agda を起動して、Emacs 上で操作を行う。

7.1 ? の使い方

Agda では、まず、ファイルを読み込み C-L (agda2-load) とすると、 全体の型がチェックされ、色で識別される (図 7.1)。

```
open import Level

postulate A : Set
postulate B : Set
postulate C : Set

postulate a : A
postulate b : A
postulate c : A

infixr 40 _::_
data  List (a) (A : Set a) : Set a where
   [] : List A
   _::_ : A → List A → List A

infixl 30 _++_
_++_ : ∀ {a} {A : Set a} → List A → List A → List A
[]       ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
          --
```

図 1　list-1

定義した式の一部を ? で置き換えると、そこに必要な型を示してくれる。

8. List の append の結合法則の証明

証明には、まず証明するべき式を型として定義する。この型を与えるλ式が証明になるのが Curry Howard 対応である。

```
list-assoc : {A : Set } → ( xs ys zs : List A ) →
  ( ( xs ++ ys ) ++ zs ) == ( xs ++ ( ys ++ zs ) )
list-assoc = ?
```

list-assoc は、入力として A の list を取る。そして、結合法則を表す等式 (を表したデータ構造 _==_) を返す。_==_ は、同じものからしか作ることはできない。

? は、

```
?0 : {A1 : Set} (xs ys zs : List A1) →
  xs ++ ys ++ zs == xs ++ (ys ++ zs)
```

という型だと Agda が教えてくれるが、これは、証明すべき型を返しているだけで証明のヒントにはならない。

List は、二つの field を持つデータ構造なので、それを場合分けすることができる。

```
list-assoc [] ys zs = ?
list-assoc (x :: xs) ys zs = ?
```

これは、Haskell のパターンマッチと同じ構文である。今度は、

```
?0 : [] ++ ys ++ zs == [] ++ (ys ++ zs)
?1 : x :: xs ++ ys ++ zs == x :: xs ++ (ys ++ zs)
```

となる。?0 の方は、_++_ の定義 ([] ++ ys = ys) から ys ++ zs になる。これは、Emacs の ?0 上にカーソルを合わせて、

```
λ ys zs → [] ++ ys ++ zs
λ ys zs → [] ++ (ys ++ zs)
```

を、それぞれ C-n (normalize) すると、両方、

```
λ ys1 zs1 → ys1 ++ zs1
```

となる。つまり両辺は同じものなので、_==_ のコンストラクタ reflection を使って生成できる。

```
list-assoc [] ys zs = reflection
```

つまり、 list-assoc [] ys zs の型を生成するのが reflection であり、list-assoc [] ys zs は ys ++ zs == ys ++ zs でもある。

```
?1 : x :: xs ++ ys ++ zs == x :: xs ++ (ys ++ zs)
```

の方は、_++_ のもう一つのパターンを使う必要がある。こちらの両辺は、C-n を使っても

```
λ x1 xs1 ys1 zs1 → x1 :: (xs1 ++ ys1 ++ zs1)
λ x1 xs1 ys1 zs1 → x1 :: (xs1 ++ (ys1 ++ zs1))
```

となり、そのままでは同じにはならない。しかし、:: の後ろの部分が証明するべき式と同じであることはわかる。List は一つ短くなっているので、これを既に証明されている式とみなして良い。[] の場合の証明は終わっているので、帰納法の基点は証明されている。つまり、後ろの部分、

```
(xs1 ++ ys1 ++ zs1) == (xs1 ++ (ys1 ++ zs1))
```

は list-assoc xs1 ys1 zs1 そのものになる。あとは、同じものに xs1 を append しても同じであることが証明できれば良い。つまり、

```
eq-cons : ∀ {n} {A : Set n} {x y : List A} ( a : A ) → x == y → ( a :: x ) == ( a :: y )
```

を証明できれば良い。

そのためには、congluence を使う。これは、同じ入力に同じ関数を作用させても同じになるという定理になる。これは、以下のように証明される。

```
cong1 : ∀ {a} {A : Set a } {b} { B : Set b } →
  ( f : List A → List B ) → {x : List A } → {y : List A} → x == y → f x == f y
cong1 f reflection = reflection
```

ここで、cong1 の最初の引数は作用させる関数 f で、二つ目の引数は x == y という式である。x == y は _==_ のデータ構造で、reflection という field を持っている。

```
cong1 f reflection
```

は、

```
list-assco [] ys zs
```

と同じパターンマッチングになる。reflection の型は x == x なので、x == y とマッチした時に、x と y は単一化されて同じものになる。同じものなので、

```
f x == f x
```

という型を reflection によって作ることができる。これで cong1 の証明ができたことになる。

f に λ x → (a :: x) を使うと eq-cons を証明できる。

```
eq-cons : ∀ {n} {A : Set n} {x y : List A} ( a : A )
  → x == y → ( a :: x ) == ( a :: y )
eq-cons a z = cong1 ( λ x → ( a :: x) ) z
```

最終的な証明は、

```
list-assoc : {A : Set } → ( xs ys zs : List A ) →
  ( ( xs ++ ys ) ++ zs ) == ( xs ++ ( ys ++ zs ) )
```

```
list-assoc [] ys zs = reflection
list-assoc (x :: xs) ys zs = eq-cons x ( list-assoc xs ys zs )
```

という形になる。残念ながら、これは人に読みやすいとは言えないが、Agda では推論規則による数式変形という形で読みやすい形にすることが可能になっている。

9. 数 式 変 形

数式変形を使うと、この証明は以下のようになる。

```
++-assoc : ∀ {n} (L : Set n) ( xs ys zs : List L ) →
 (xs ++ ys) ++ zs == xs ++ (ys ++ zs)
++-assoc A [] ys zs = let open ==-Reasoning A in
 begin – to prove ([] ++ ys) ++ zs == [] ++ (ys ++ zs)
 ( [] ++ ys ) ++ zs
 ==⟨ reflection ⟩
  ys ++ zs
 ==⟨⟩
  [] ++ ( ys ++ zs )
 ∎
++-assoc A (x :: xs) ys zs = let open ==-Reasoning A in
 begin
  ((x :: xs) ++ ys) ++ zs
 ==⟨⟩
  (x :: (xs ++ ys)) ++ zs
 ==⟨⟩
  x :: ((xs ++ ys) ++ zs)
 ==⟨ cong1 (_::_ x) (++-assoc A xs ys zs) ⟩
  x :: (xs ++ (ys ++ zs))
 ==⟨⟩
  (x :: xs) ++ (ys ++ zs)
 ∎
```

数式が ==⟨ cong1 (_::_ x) (++-assoc A xs ys zs) ⟩ などで変形されている。 ==⟨ reflection ⟩ は、何の変形もしてないが、両方が異なる見かけの同じ正規型である時に見かけを変形している。 ==⟨⟩ と書くこともできる。

9.1 数式変形の記述

この数式変形自体も Agda で記述されている。

```
module ==-Reasoning {n} (A : Set n ) where
 infixr 2 _∎
 infixr 2 _==⟨_⟩_ _==⟨⟩_
 infix 1 begin_
 data _IsRelatedTo_ (x y : List A) :
        Set n where
  relTo : (x≈y : x == y ) → x IsRelatedTo y
 begin_ : {x : List A } {y : List A} →
    x IsRelatedTo y → x == y
 begin relTo x≈y = x≈y
 _==⟨_⟩_ : (x : List A ) {y z : List A} →
    x == y → y IsRelatedTo z → x IsRelatedTo z
 _ ==⟨ x≈y ⟩ relTo y≈z = relTo (trans-list x≈y y≈z)
 _==⟨⟩_ : (x : List A ) {y : List A}
    → x IsRelatedTo y → x IsRelatedTo y
 _ ==⟨⟩ x≈y = x≈y
 _∎ : (x : List A ) → x IsRelatedTo x
 _∎ _ = relTo reflection
```

これは List 専用の式変形を定義している。trans-list は、List での等式の三段論法になっている。これは、単一化で証明される。

```
trans-list : ∀ {n} {A : Set n} {x y z : List A}
  → x == y → y == z → x == z
trans-list reflection reflection = reflection
```

relTo が、等しくなるべき等式を持っていて、証明の最後を示す _∎ _ が、三段論法で繋がったすべての等式が等しいことを要求する。最後に　begin_ が _IsRelatedTo_ を _==_ に変換する。

数式変形は任意の等式に使えるが、自分用にカスタマイズして使う方が良い。良く使用する推論や定理も一緒に記述する方が良い。

Agda の module には引数があり、暗黙の引数も使うことができる。

10. record の使い方

data は constructor を提供するが、record は数学的構造の存在を示す。以下は[10] による圏の定義である。

```
record IsCategory {c1 c2 ℓ : Level} (Obj : Set c1)
  (Hom : Obj → Obj → Set c2)
  (_≈_ : {A B : Obj} → Rel (Hom A B) ℓ)
  (_o_ : {A B C : Obj} → Hom B C → Hom A B → Hom A C)
  (Id : {A : Obj} → Hom A A) : Set (suc (c1 ⊔ c2 ⊔ ℓ)) where
 field
  isEquivalence : {A B : Obj} →
    IsEquivalence {c2} {ℓ} {Hom A B} _≈_
  identityL : {A B : Obj} → {f : Hom A B} → (Id o f) ≈ f
  identityR : {A B : Obj} → {f : Hom A B} → (f o Id) ≈ f
  o-resp-≈ : {A B C : Obj} {f g : Hom A B} {h i : Hom B C}
     → f ≈ g → h ≈ i → (h o f) ≈ (i o g)
  associative : {A B C D : Obj} {f : Hom C D}
    {g : Hom B C} {h : Hom A B}
    → (f o (g o h)) ≈ ((f o g) o h)
```

ここでは、圏の要素がコンストラクタ IsCategory に列挙されている。そして、圏が満たす性質 (公理) が field に列挙されている。IsEquivalence は、library で定義されているものを呼び出している。

field は data 型と異なり、要素へアクセスするア

クセサを提供している。なので、record field は、パターンには使えない。圏を作るときには、アクセサをすべて定義する必要がある。つまり、圏であることを示すには公理をすべて満たすことを示す必要がある。例えば、

```
isKleisliCategory : IsCategory ( Obj A ) KHom _≈̃_ _*_ K-id
isKleisliCategory = record { isEquivalence = isEquivalence
  ; identityL = KidL
  ; identityR = KidR
  ; o-resp-≈ = Ko-resp
  ; associative = Kassoc
  }
```

のように record の中で filed を全部定義する。(KidL などは、別に定義されているとする)

record を入力に使うと、filed は、すべて仮定される。つまり、filed に記述した性質を持つ数学的存在を仮定することになる。なので、record は、数学的存在を記述するのに向いている。

ここで、Hom は圏の arrow であり、対象を二つ引数とし何かの集合を返す関数として定義されている。_≈̃_ は、arrow の等号だが、data 型としては定義されていなくて、Rel (Hom A B) ℓ　となっている。これは、Relation.Binary.Core に定義されていて、

```
Rel : ∀ {a} → Set a → (ℓ : Level) → Set (a ⊔ suc ℓ)
Rel A ℓ = A → A → Set ℓ
```

となっている。つまり arrow と同じように集合で定義されている。ここでは A という型を持つ引数を二つ持ち、何かの 集合を返す関数である。このせいで、arrow には _≡̇_ を使うことができない。合同性 (同じものに同じ関数を作用させても同じ) は、 _≡̇_ を仮定しているので、この圏の定義では、合同性を自由に使うことはできない。もし、arrow が Set であれば、あるいは、_≈̃_ → _≡̇_ を仮定すれば、 _≡̇_ を使うことができる。

11. infix operator

infix operator には任意の Unicode を使うことができ、数学らしい記号を使うことが可能になっている。これは、Agda が monomorphism な言語であるために、型の組み合わせ毎に別な名前を用意する必要がある状況で特に便利になっている。

```
  associative : {A B C D : Obj} {f : Hom C D} {g : Hom B C} {h
: Hom A B}
              → (f o (g o h)) ≈ ((f o g) o h)
```

では、_o_ が infix operator であり、_ の部分が引数の構文的に置かれる場所を示している。しかし、_o_ は、record 内部で定義されていて、record の parameter を引数と持っている。実際、Category._o_ は、

```
λ a b c → Category._o_
```

という型を持っていて、引数を三つ持っている。最初の引数は record であり、次の二つが、Hom (arrow) である。なので、折角、infix operator を定義しても、record の外で使う時には、引数がずれてしまう。そこで、

```
_[_o_] : ∀ {c₁ c₂ ℓ} → (C : Category c₁ c₂ ℓ) → {a b c : Obj C} →
Hom C b c → Hom C a b → Hom C a c
C [ f o g ] = Category._o_ C f g
```

などと定義することがある。これで、圏 C の中での射の演算であることがわかり、表示も綺麗になる。

この圏 C は、module parameter にすると、暗黙の引数として使うことができ、そこで import Category すれば、_o_ を本来の infix operator として再定義して、使用することができる。

12. record での変数の位置の選択

record 型を入力で使う場合は、record の引数は、record で定義される数学的構造を定義した時に決まっている必要がある。field に書かるのは関数か関数で表された式になるが、これは、record が勝手に提示するものになり、外から指定することはできない。

逆に record を値として定義する時には、record の引数は指定できず、field の値のみを指定できる。record の引数は field の中の式の型から決まる。

関手を record で定義する時に、二つの射 (対象の写像 FObj と射の写像 FMap) は field として定義する。record IsFunctor では関手の入力と出力に対応する二つの圏は record の引数になっている。一方で、record Functor では関手の入力と出力に対応する二つの圏は record の field になっている。このように二段階で数学的構造を定義すると、値としての field と、公理としての field を分離することができる。

値は、入力となる record の引数に記述すると「すべての、その型の値に対して record が存在する」という意味になり、入力となる record の field に記述すると、「record は、その型の値を生成する」になる。なので、どちらに記述するかの差は大きい。

```
record IsFunctor {c₁ c₂ ℓ c₁′ c₂′ ℓ′ : Level}
  (C : Category c₁ c₂ ℓ) (D : Category c₁′ c₂′ ℓ′)
  (FObj : Obj C → Obj D)
  (FMap : {A B : Obj C} → Hom C A B → Hom D (FObj A) (FObj
B))
  : Set (suc (c₁ ⊔ c₂ ⊔ ℓ ⊔ c₁′ ⊔ c₂′ ⊔ ℓ′)) where
 field
  ≈-cong : {A B : Obj C} {f g : Hom C A B} →
    C [ f ≈ g ] → D [ FMap f ≈ FMap g ]
```

```
  identity : {A : Obj C} → D [
    (FMap {A} {A} (Id {_} {_} {_} {C} A)) ≈ (Id {_} {_} {_} {D}
(FObj A)) ]
  distr : {a b c : Obj C} {f : Hom C a b} {g : Hom C b c}
   → D [ FMap (C [ g o f ]) ≈ (D [ FMap g o FMap f ] )]
record Functor {c1 c2 ℓ c1′ c2′ ℓ′ : Level}
  (domain : Category c1 c2 ℓ) (codomain : Category c1′ c2′ ℓ′ )
 : Set (suc (c1 ⊔ c2 ⊔ ℓ ⊔ c1′ ⊔ c2′ ⊔ ℓ′ )) where
 field
  FObj : Obj domain → Obj codomain
  FMap : {A B : Obj domain} → Hom domain A B → Hom codomain
(FObj A) (FObj B)
  isFunctor : IsFunctor domain codomain FObj FMap
```

13. Agda では証明できないもの

一連の証明で、まだ証明できない部分は、postulate を使って仮定してしまうという方法がある。また、証明できてない部分を入力に明示的に記述しても良い。? で放置すると、そこの部分がいつも表示される欠点がある。

逆に、Agda が証明が終わったと表示 (何もエラーを表示しない) としても、これらの証明されてない部分が残っていることがある。

Agad は直観主義論理であり、証明できない式がいくつかあるが、それらは仮定しても構わない。そのような命題の一つは、 以下の関数外延性である。

```
Extensionality a b = {A : Set a} {B : A → Set b} {f g : (x : A) →
B x}
 → (∀ x → f x ≡ g x) → f ≡ g → ( λ x → f x ≡ λ x → g x )
```

つまり、全ての値について等しいからと言って、その関数自体が (_≡_ の意味で) 等しいということを証明することはできない。これは、仮定する必要がある。関数が仮定ではなく実際に構成されているものならば、この仮定は必要ない。

この式は、Agda のライブラリに記述されていて、必要ならばそれを postulate すれば良い。

Rel で定義された等式から、_≡_ を導くこともできない。もし、圏の射が集合ならば射の等式は _≡_ で定義されるべきなので、以下を仮定しても良い、あるいはする必要がある。

```
postulate ≈-≡ : { c1 c2 ℓ : Level} { A : Category c1 c2 ℓ }
 {a b : Obj A } { x y : Hom A a b } →
  (x≈y : A [ x ≈ y ]) → x ≡ y
```

米田レンマは射が集合であることを要求するので、明示的に集合全体の圏に対して証明するか、任意の圏に対して、これと関数外延性を仮定して証明する必要がある。

また、集合以外の関係に対する合同性も単独では証明できない。例えば、関手の合同性は、自明には成り立たないので、以下のように IsFunctor の公理とする必要がある。

```
≈-cong : {A B : Obj C} {f g : Hom C A B}
  → C [ f ≈ g ] → D [ FMap f ≈ FMap g ]
```

また、

```
a b : Set
```

の場合に、a ≡ b は証明できない。 _≡_ は、初めから同じもの (x ≡ x) に対して生成できるので、最初から異なる可能性のあるものには成立しない。合同性の逆、写像した先で等しければ、元も等しいは、例えば以下の米田関手の Full embededing は、

```
FObj YonedaFunctor a ≡ FObj YonedaFunctor b → a ≡ b
```

この形で証明することはできない。逆写像を実際に構築することによってしか証明できない。

14. Agda でのデバッグ

Agda は実行することが目的の言語ではないので、デバッグはコンパイル時エラーに対してのみ行う。主なエラーは三種類である。

最初は構文エラーである。Agda は、ほとんどの場合、語の区切りに空白を要求する。特に → や =: の両側に空白を要求する。例外は () と のみである。空白がないと一つの語だと解釈される。x=y は x=y という一つの変数である。これは、式を変数に格納する時には読みやすい。また、さまざまな Unicode の記号を自由に使えるという利点がある。しかし、実際には、構文エラーの大半は空白の入れ忘れである。

もう一つのエラーは型の不整合である。これは、Emacs 上で赤で表示される。不整合の部分を ?に置き換えれば、そこにあるべきものの型が示されるので、それを入れれば良い。これは比較的簡単に解決されるエラーである。

最後は、変数が十分に代入されてない場合のエラーである。record の field が定義されてないのに、そこへの参照が必要な場合などに、その部分が黄色で表示される。これは、record の深い部分で起こることもあり、解決が難しい。

この場合に、まずすることは「 暗黙の変数を明示」である。これにより解決することがある。関数の呼び出し時に、a などとして指定する。

同じものを繰り返し記述することは、普通のプログラムでも変更を複数の場所で行うことになるので望ましくない。Agda でも同じことが言える。そのために Agda には module がある。

module には parameter があり、それを module 中

で暗黙の入力として使うことができる。一方で、証明するべきに式に明示的あるいは暗黙の引数として記述しても良い。どちらにするかは、 Agda の Monomorphism から来る問題がある。

Agda は一つのファイルの中で、異なる型の持つ関数は異なる名前である必要がある。型が異なる同じ名前の関数を作ることはできない。これは、module を異なる型で複数回呼び出すことができないということを意味する。module parameter ではなく、個々の式の入力すれば、module を複数回呼び出す必要はなくなる。

暗黙の変数は便利であるが、増やすと Agda のメモリを消費し、証明のチェックが遅くなる。その場合は、明示的な変数に変えるとなおる場合がある。

15. 圏 論 の 例

ここまでで、 圏の定義と 関手の定義を使ってきた。自然変換の定義は、以下のようになる。

```
record IsNTrans {c1 c2 ℓ c1′ c2′ ℓ′ : Level}
  (D : Category c1 c2 ℓ) (C : Category c1′ c2′ ℓ′ )
  ( F G : Functor D C )
  (TMap : (A : Obj D) → Hom C (FObj F A) (FObj G A))
      : Set (suc (c1 ⊔ c2 ⊔ ℓ ⊔ c1′ ⊔ c2′ ⊔ ℓ′ )) where
 field
  commute : {a b : Obj D} {f : Hom D a b}
  → C [ C [ ( FMap G f ) o ( TMap a ) ] ≈ C [ (TMap b ) o (FMap F f) ] ]
record NTrans {c1 c2 ℓ c1′ c2′ ℓ′ : Level}
   (domain : Category c1 c2 ℓ) (codomain : Category c1′ c2′ ℓ′ )
   (F G : Functor domain codomain )
   : Set (suc (c1 ⊔ c2 ⊔ ℓ ⊔ c1′ ⊔ c2′ ⊔ ℓ′ )) where
 field
  TMap : (A : Obj domain) → Hom codomain (FObj F A) (FObj G A)
  isNTrans : IsNTrans domain codomain F G TMap
```

これは、以下のような可換図に対応する

$$\begin{array}{ccc} F(a) & \xrightarrow{F(f)} & F(b) \\ \downarrow{\scriptstyle t(a)} & & \downarrow{\scriptstyle t(b)} \\ G(a) & \xrightarrow[G(f)]{} & G(b) \end{array}$$

今までと同じように二つにわけて定義している。

これを使って、 Monad の結合性の証明を以下のように行うことができる。まず Monad を定義する。

```
record IsMonad {c1 c2 ℓ : Level} (A : Category c1 c2 ℓ)
       ( T : Functor A A )
       ( η : NTrans A A identityFunctor T )
       ( μ : NTrans A A (T ○ T) T)
       : Set (suc (c1 ⊔ c2 ⊔ ℓ )) where
 field
  assoc : {a : Obj A} → A [ A [ TMap μ a o TMap μ ( FObj T a ) ]
      ≈ A [ TMap μ a o FMap T (TMap μ a) ] ]
  unity1 : {a : Obj A} → A [ A [ TMap μ a o TMap η ( FObj T a ) ]
      ≈ Id {_} {_} {_} {A} (FObj T a) ]
  unity2 : {a : Obj A} → A [ A [ TMap μ a o (FMap T (TMap η a ))]
      ≈ Id {_} {_} {_} {A} (FObj T a) ]
record Monad {c1 c2 ℓ : Level} (A : Category c1 c2 ℓ)
     (T : Functor A A) (η : NTrans A A identityFunctor T)
     (μ : NTrans A A (T ○ T) T)
    : Set (suc (c1 ⊔ c2 ⊔ ℓ )) where
 field
  isMonad : IsMonad A T η μ
   – g ○ f = μ (c) T(g) f
  join : { a b : Obj A } → { c : Obj A } →
   ( Hom A b ( FObj T c )) → ( Hom A a ( FObj T b)) → Hom A a ( FObj T c )
  join {_} {_} {c} g f = A [ TMap μ c o A [ FMap T g o f ] ]
```

Id の暗黙の引数は、すべて指定する必要がある。証明するべき式は、

```
join M h (join M g f) ≈ join M ( join M h g) f
```

である。この証明は以下のようになる。

```
– h ○ (g ○ f) = (h ○ g) ○ f
Lemma9 : { a b c d : Obj A }
       ( h : Hom A c ( FObj T d) )
       ( g : Hom A b ( FObj T c) )
       ( f : Hom A a ( FObj T b) )
         → A [ join M h (join M g f) ≈ join M ( join M h g) f ]
Lemma9 {a} {b} {c} {d} h g f =
 begin
  join M h (join M g f)
  ≈⟨⟩
  join M h ( ( TMap μ c o ( FMap T g o f ) ) )
  ≈⟨⟩
  ( TMap μ d o ( FMap T h o ( TMap μ c o ( FMap T g o f ) ) ) )
  ≈⟨ cdr ( cdr ( assoc )) ⟩
  ( TMap μ d o ( FMap T h o ( ( TMap μ c o FMap T g ) o f ) ) )
  ≈⟨ assoc ⟩ — ( f o ( g o h ) ) = ( ( f o g ) o h )
  ( ( TMap μ d o FMap T h ) o ( (TMap μ c o FMap T g ) o f ) )
  ≈⟨ assoc ⟩
  ( ( ( TMap μ d o FMap T h ) o (TMap μ c o FMap T g ) ) o f )
  ≈ ↑ ⟨ car assoc ⟩
  ( ( TMap μ d o ( FMap T h o ( TMap μ c o FMap T g ) ) ) o f )
  ≈⟨ car ( cdr (assoc) ) ⟩
  ( ( TMap μ d o ( ( FMap T h o TMap μ c ) o FMap T g ) ) o f )
  ≈⟨ car assoc ⟩
```

```
  ( ( ( TMap μ d o ( FMap T h o TMap μ c ) ) o FMap T g ) o f )
  ≈⟨ car (car ( cdr ( begin
    ( FMap T h o TMap μ c )
   ≈⟨ nat μ ⟩
    ( TMap μ (FObj T d) o FMap T (FMap T h) )
   ∎
  ))) ⟩
  ( ( ( TMap μ d o ( TMap μ ( FObj T d) o FMap T ( FMap T h
) ) ) o FMap T g ) o f )
  ≈ ↑ ⟨ car assoc ⟩
  ( ( TMap μ d o ( ( TMap μ ( FObj T d) o FMap T ( FMap T h
) ) o FMap T g ) ) o f )
  ≈ ↑ ⟨ car ( cdr assoc ) ⟩
  ( ( TMap μ d o ( TMap μ ( FObj T d) o ( FMap T ( FMap T h
) o FMap T g ) ) ) o f )
  ≈ ↑ ⟨ car ( cdr (cdr (distr T ))) ⟩
  ( ( TMap μ d o ( TMap μ ( FObj T d) o FMap T ( ( FMap T h
o g ) ) ) ) o f )
  ≈⟨ car assoc ⟩
  ( ( ( TMap μ d o TMap μ ( FObj T d) ) o FMap T ( ( FMap T
h o g ) ) ) o f )
  ≈⟨ car ( car (
   begin
    ( TMap μ d o TMap μ (FObj T d) )
   ≈⟨ IsMonad.assoc ( isMonad M) ⟩
    ( TMap μ d o FMap T (TMap μ d) )
   ∎
  )) ⟩
  ( ( ( TMap μ d o FMap T ( TMap μ d) ) o FMap T ( ( FMap T
h o g ) ) ) o f )
  ≈ ↑ ⟨ car assoc ⟩
  ( ( TMap μ d o ( FMap T ( TMap μ d ) o FMap T ( ( FMap T
h o g ) ) ) ) o f )
  ≈ ↑ ⟨ assoc ⟩
  ( TMap μ d o ( ( FMap T ( TMap μ d ) o FMap T ( ( FMap T
h o g ) ) ) o f ) )
  ≈ ↑ ⟨ cdr ( car ( distr T )) ⟩
  ( TMap μ d o ( FMap T ( ( ( TMap μ d ) o ( FMap T h o g ) ) )
o f ) )
  ≈⟨⟩
  join M ( ( TMap μ d o ( FMap T h o g ) ) ) f
  ≈⟨⟩
  join M ( join M h g) f
 ∎ where open ≈-Reasoning (A)
```

ここで、assoc は射の合成の結合法則を使って、式を変形していることを示している。nat μ は、自然変換μの commute 規則を呼び出している。≈ ↑ ⟨⟩ は、逆向きの推論、≈⟨⟩ は正規型どおしが同じ場合の見かけだけの変換である。 distr T は関手 T の分配法則である。car と cdr は、括弧のレベルの深い部分に推論規則を適用することを示す。begin/∎ の式変形は入れ子にすることができて、式の特定の部分に対して、着目して変形することできる。

これらの定義は以下のようになっている。

```
module ≈-Reasoning {c₁ c₂ ℓ : Level} (A : Category c₁ c₂ ℓ) where
  _o_ : {a b c : Obj A } ( x : Hom A a b ) ( y : Hom A c a ) → Hom
A c b
  x o y = A [ x o y ]
  _≈_ : {a b : Obj A } → Rel (Hom A a b) ℓ
  x ≈ y = A [ x ≈ y ]
  infixr 9 _o_
  infix 4 _≈_
  refl-hom : {a b : Obj A } { x : Hom A a b } → x ≈ x
  refl-hom = IsEquivalence.refl
   (IsCategory.isEquivalence ( Category.isCategory A ))
  car : {a b c : Obj A } {x y : Hom A a b } { f : Hom A c a } →
     x ≈ y → ( x o f ) ≈ ( y o f )
  car {f} eq = ( IsCategory.o-resp-≈ ( Category.isCategory A ))
    ( refl-hom ) eq
  cdr : {a b c : Obj A } {x y : Hom A a b } { f : Hom A b c } →
     x ≈ y → f o x ≈ f o y
  cdr {f} eq = ( IsCategory.o-resp-≈ ( Category.isCategory A ))
    eq (refl-hom )
  assoc : {a b c d : Obj A }
    {f : Hom A c d} {g : Hom A b c} {h : Hom A a b}
              → f o ( g o h ) ≈ ( f o g ) o h
  assoc = IsCategory.associative (Category.isCategory A)
  distr : { c₁ c₂ ℓ : Level} {A : Category c₁ c₂ ℓ}
    { c₁′ c₂′ ℓ′ : Level} {D : Category c₁′ c₂′ ℓ′ }
   (T : Functor D A) → {a b c : Obj D} {g : Hom D b c} { f : Hom
D a b }
    → A [ FMap T ( D [ g o f ] ) ≈ A [ FMap T g o FMap T f ] ]
  distr T = IsFunctor.distr ( isFunctor T )
  open NTrans
  nat : { c₁ c₂ ℓ : Level} {A : Category c₁ c₂ ℓ}
   { c₁′ c₂′ ℓ′ : Level} {D : Category c₁′ c₂′ ℓ′ }
    {a b : Obj D} {f : Hom D a b} {F G : Functor D A }
   → (η : NTrans D A F G )
   → A [ A [ FMap G f o TMap η a ] ≈ A [ TMap η b o FMap F f ]
]
  nat η = IsNTrans.commute ( isNTrans η )
```

IsMonad.assoc (isMonad M) は、Monad の結合則が使われている部分である。

この証明は、もちろん手での証明、清書された証明と同一である。、この証明を見ると、Monad の規則にうち、結合則のみを使っていることがわかる。自然変換の可換則を一箇所、T の分配則を二箇所で使っていることがわかる。

手書きの式に比べるとかなり長いが、そこでの式の対応は Agda 言語にそって決まっている。また、証

明を構築していく時に、? を使った単一化が道標となる。式の変形では、ad-hoc な記号が使われる。実際、T は、射の関数と対象の関数で同じ T が使われているが、それは別な式になっている。これになれるには、それなりの演習を解く必要があるが、Agda では FMap, FObj などで明示的に区別されている。

16. Agda の有効性

Agda は Haskell の構文との親和性があり、Haskell を知っていれば、他の証明支援系よりも学びやすいと思われる。

本や自分の証明と異なり、証明の完成度がわかる。逆に、ほとんどすべてを証明しないと証明が完結しない。

推論規則や式を見ることができるので、証明の対称性を視認できる。特に式変形を使うことより、証明の対称性を可視化できる。

? による単一化 により証明を見つけることができる。しかし、Agda は自動化する方法がほとんど存在しないので、すべて人手で証明する必要がある。

Monomoprhism があり、勝手な記号を導入することが広く行われている。さらに、record 定義や module 定義に任意性があり、 証明は極めて個人的なものになる。module をそのまま使う以外に証明を共有することは難しい。式変形ではなく、λ式により直接証明を記述する場合は、Coq と同様に視認性の低い証明になってしまう。

Agda は特に圏論の証明に向いており、 Monad や Kan extension を「証明として」理解するためには有効であると思われる。しかし、証明を理解することは数学的に理解することとは異なる。実際、モデルを理解することとの差がある。圏論の場合は、モデルはグラフであり、モデル的に理解することと証明から理解することのギャップは小さい。

例えば、自然変換の可換性は、純粋に記号的な可換性であり、そのパターンを理解することが自然変換を理解することになる。

自分で Agda で証明を構築することは、一種のパズル、あるいは詰将棋のようなものであり、それ自体が知的遊戯としてすぐれている。これは、省略されている数学書を補う仕組みになっている。実際、Agda のようなツール抜きでも、数式を手書きするなどの方法で、それを補うことをしなければ、数学を理解することはできない。Agda は、その際に起きるエラーを指摘してくれるので、未熟な理解を防ぐことができる。

証明の対称性と、パターンは、それ自体が数学的証明の技術であり、Agda を使うことにより、その技術を身につけられることが大きい。自然変換の可換性を可換図で記述することは広く使われているが、Agda で記述すると、それは、関手と自然変換の式のパターンとなる。

Monad を理解するのに、join の結合性を理解することが必須かというと、そのようなことはない。むしろ、それが重要であることを主張している解説は少ない。しかし、

- T が関手 (データ構造)
- return が自然変換η (プログラム)
- join が自然変換μ (プログラム)

という対応を理解しなければ、Monad を使う理由を理解できない。実際、Monad は対象レベルとメタレベルの双対性 (随伴関手) が反映されたものになっている。

17. 今後の展開

Agda は、Emacs に密接に結びついた会話的な証明支援系だが、実際のプログラミングに比べると、一行当たりにかかる時間が十倍以上ある。1 日にかける行数は 100 行が限界に近い。難しい部分では一行に数日かかることもある。

圏論を理解するのに Agda が有効であることはわかったが、これを実用的な (例えば)Haskell のプログラムに応用するには、人間的、あるいは計算環境的なギャップがある。

例えば、Monad は、その組み合わせを Monad にすることが難しいことが知られている。Monad の組み合わせ自体は、Haskell で作れるが、組み合わせた join が、元の join と、どう関係あるかを示すのは難しい。それを示すには Agda などのシステムが役に立つかもれない。

今後は、さらに Agda での圏論の理解を深めると共に、実用的なプログラムの信頼性を高めるのに Agda (あるいは、これに類似したシステム) が使えるかどうかを調べていく予定である。

参 考 文 献

1) Kinoshita, Y.: Agda language, Technical Report PS-2008-014, 独立行政法人産業技術総合研究所 (2008).
2) O'Sullivan, B., Stewart, D. and Goerzen, J.: Real World Haskell (2008).
3) Girard, J.-Y., Taylor, P. and Lafont, Y.: *Proofs and Types*, Cambridge University Press, New York, NY, USA (1989).
4) Gordon, M. and Melham, T.: Introduction to the HOL System, Technical report, Cambridge University (1994).
5) Moggi, E.: Computational Lambda-calculus and Monads, *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, Piscataway, NJ, USA, IEEE Press, pp. 14–23 (1989).

6) Lambek, J. and Scott, P. J.: *Introduction to Higher Order Categorical Logic*, Cambridge University Press, New York, NY, USA (1986).
7) Burch, J., Clarke, E., McMillan, K., Dill, D. and Hwang, J.: Symbolic model checking: 10^{20} states and beyondg, *Proc. of the Fifth Anual IEEE Symposium on Logic in Computer Science* (1990).
8) Wiedijk, F.: *The Seventeen Provers of the World: Foreword by Dana S. Scott (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence)*, Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006).
9) KONO, S.: https://github.com/shinji-kono/category-agda.git (2013).
10) Ishii, H.: https://github.com/konn/category-agda.git (2011).