邦訳：How to make ad-hoc polymorphism less ad hoc

アドホック多相をより「場当たり的 (ad hoc)」でなくするには

Philip Wadler and Stephen Blott, University of Glasgow, October 1988.

この記事について

訳した日： 2018 年 12 月 15 日、訳した人： @unnohideyuki[*1]

これは、Haskell Advent Calendar 2018[*2] 15 日めの記事です。型クラスについて書かれた論文である、How to make ad-hoc polymorphism less ad hoc を訳してみました[*3]。面白い論文なので、一人でも多くの方に読まれるといいなと思います。

概要

本稿は、いわゆるアドホック多相への新しいアプローチである型クラスを提案する。型クラスは、乗算のような算術演算子の多重定義を可能にし、また、Standard ML における等値型（eqtype）変数を一般化する。型クラスは、Hindley/Milner の多相型システムを拡張したものであり、オブジェクト指向プログラミング、有界な量化や、抽象データ型における問題への新しいアプローチを提供する。本稿では、まず、型クラスをインフォーマルに導入したのちに、型推論規則を用いて形式的な定義を与える。

1. はじめに

Strachey は、二種類の多相を区別するために「アドホック」と「パラメトリック」という形容詞を選んだ [Str67]。

アドホック多相は、ある関数がいくつかの異なる型に対して定義され、それぞれの型に応じて異なる動作をするときに起こる。典型的な例は、多重定義された乗算である： たとえば、整数の乗算 (3*3 など）と浮動小数点数の乗算（3.14*3.14）に同じ記号を用いることは、しばしば行われている。

パラメトリック多相は、ある関数がある範囲の型に対して定義され、いずれの型に対しても同じ動作をするときに生じる。典型的な例が range 関数である。これは、整数のリストに対しても、浮動小数点数のリストに対しても同じように動作する。

パラメトリック多相に対して広く受け入れられているアプローチの一つが、Hindley/Milner の型システム [Hin69, Mil78, DM82] であり、これは、Standard ML, Miranda 他の言語で採用されている。その一方で、アドホック多相に対する広く受け入れられたやり方はなく、そういう意味でも「アドホック」と呼ぶにふさわしいといえる。

本稿は型クラスを提案する。これは、ある種の多重定義も扱えるように Hindley/Milner の型システムを拡張し、Strachey が二つに分けた多相をひとまとめにする。

ここで示される型システムは、Hindley/Milner の型システムを一般化したものである。Hindley/Milner 型

[*1] https://twitter.com/unnohideyuki

[*2] https://qiita.com/advent-calendar/2018/haskell

[*3] 付録がまだですが、きっと今年中に！

システムにおけるのと同様、型宣言は推論され、関数の明示的な型宣言は必須ではない。推論の過程で、型クラスを用いたプログラムは、多重定義を用いていない等価なプログラムに変換することが出来る。変換後のプログラムは、（一般化されていない、元々の）Hindely/Milner 型システムによって型付け可能である。

本稿の本文では、型クラスと翻訳規則への形式でない概論を述べ、付録では、型付けと翻訳の形式的な規則を、推論規則の形（[DM82] におけるのと同様）で与える。翻訳規則は、型クラスの意味論を与える。翻訳規則は、また、実現できる実装技術をひとつ示しているともいえる：望むならば、単にプリプロセッサを書くことで、既存の Hindly/Milner 型の言語に新たなシステムを追加することも可能である。

アドホック多相の問題が生じる箇所は 2 つあり、それは、算術演算子と同値の定義である。以下では、これら 3 つの問題[*4]に対するアプローチとして、Standard ML や Miranda で採用されたものを調べる。これらのアプローチは、言語間で異なるだけでなく、同じ言語のなかでさえ異なっている。しかし、後にみるように、型クラスはこれら 3 つの問題への統一的な機構を提供する。

この研究は、Haskell 委員会が遅延評価の関数型プログラミング言語を設計するにあたって検討したものを元にしている。Haskell 委員会の目標のひとつは、様々な問題に対して、可能なかぎり「既成の」解決方法を適用することだった。なので、我々は、算術と等値に対する標準的に使える解決方法がないと判明したことには、少々驚かされた！型クラスは、これらの問題に対するより良い解決を探す試みとして開発され、Haskell の言語設計に取り入れられる程度には成功したとみなされている。しかし、型クラスは、Haskell とは独立に判断されるべきものであって、Standard ML といった他の言語に組み込むことも可能である。

型クラスは、オブジェクト指向プログラミング、型の有界量化や、抽象データ型における問題と密接に関連しているようである。そういった関連について、いくらかは以下に述べるが、きちんと理解するためには他書に頼る必要があるだろう。

我々のものに似た型システムは、Stefan Kaes [Kae88] によっても独立に発見されている。我々のものは、いくつかの点で Kaes のものを改善している。特に、関連する演算子をグループわけするための型クラスの導入、および、より良い翻訳手法の導入においてである。

本稿は、二つの部分から成る: 本文では、型クラスについての形式的でない導入をし、付録では、より形式的な記述を与える。第 2 節では、Standard ML や Miranda で用いられるアドホック多相の限界について述べることで、新しいシステムへの動機づけをする。第 3 節では、単純な例を用いて型クラスを導入する。第 4 節では、第 3 節で述べた例を、型クラスのない等価なプログラムに翻訳する方法を示す。第 5 節では、もう一つの例として、多重定義された等価演算の定義を述べる。第 6 節では、サブクラスについて述べ、第 7 節においては、関連研究と結論を述べる。付録Aでは、型付けと翻訳の推論規則を示す。

2. アドホック多相の限界

アドホック多相の取り扱いに関する動機づけのため、この節では、Standard ML と Miranda における算術および等値で生じる問題を調べる。

算術

多重定義の最も単純なアプローチでは、加算や乗算のような基本演算は多重定義されるものの、それらを用いた関数は多重定義されない。例えば、3*3 や 3.14*3.14 とは書けるが、以下のように定義しておいて、

[*4] 段落の冒頭では 2 つと言っているのになぁ

```
square x = x * x
```

さらに、次のような式を書くことは **できない**

```
square 3
square 3.14
```

これは、Standard ML で採られていたアプローチである。(ちなみに、Standard ML は多重定義された算術演算子を持つにもかかわらず、面白いことに、形式的定義では多重定義の解決は故意に曖昧になっていて [HMT88, page 71]、異なるバージョンの Standard ML は、異なる方法で多重定義を解決する。)

より一般的なアプローチでは、上に示した等式が、多重定義された 2 つの版 (Int -> Int および Float -> Float) の square 関数定義として許容される。しかし、次のような関数を考えてみよう :

```
squares (x, y, z)
  = (square x, square y, square z)
```

ここでは x, y および z は独立に Int 型か Float 型のいずれかをとり得るため、この関数には、多重定義された版が 8 つ生じることになる。一般に、このような変換の個数は指数関数的に増大し、これが、このアプローチが広く用いられていない理由のひとつとなっている。

Miranda は、算術演算を多重定義しないことで、この問題を「すり抜け」ている。Miranda には、(”num” と呼ばれる) 浮動小数点型しかなく、したがって、演算が整数に限られることを型システムを用いて示すことは出来ない。

等値

等値演算子の歴史はバリエーションに富んでいる。等値は、多重定義として扱われたり、全面的に多相として扱われたり、部分的に多相として扱われたりしてきた。

最初のアプローチは、乗算と同じように、等値を多重定義する。具体的には、等値演算が、等値を認める単一型 (つまり、抽象型や関数型はこれに含まれない) 全てに対して多重定義される。このような言語では、整数同士の比較の意味で 3*4 == 12 と書いたり、文字同士の比較の意味で 'a' == 'b' のように書くことができる。しかし、等値ももちいて以下のような member 関数

```
member [] y = False
member (x:xs) y = (x == y) \/ member xs y
```

を書いて、そして、次のように用いることは出来ない

```
member [1, 2, 3] 2
member "Haskell" 'k'
```

(本稿では、文字列のリスト ['a', 'b', 'c'] を "abc" のように省略する。) このアプローチは、最初のバージョンの Standard ML で用いられた [Mil84]。

第二のアプローチは、等値を全面的に多相にする。この場合、等値の型は以下のようになる：

```
(==) :: a -> a -> Bool
```

ここで、a は全ての型の範囲をとる型変数である。このアプローチにおいて、member の型は次のようになる：

```
member :: [a] -> a -> Bool
```

([a] は「a のリスト」の型という意味。) これの意味するところは、関数や抽象型に等値を適用しても型エラーにはならないということである。このアプローチは Miranda で採用された。もし関数に等値を適用した場合にはランタイムエラーとなり、また、抽象データ型に等値を適用したときには、結果は、データ表現同士の比較となる。これは、抽象の原理を違反しており、バグの原因となり得る。

第三のアプローチは、等値を、ある限られたやり方で多相にする。この場合、等値の型は以下のようになる：

```
(==) :: a(==) -> a(==) -> Bool
```

ここで、a(==) は、等値を許す型の範囲をとる型変数である。今後は、member の型は次のようになる：

```
member :: [a(==)] -> a(==) -> Bool
```

また、等値や member を関数や抽象データ型に適用すると、型エラーとなる。これは、現在 Standard ML が採用しているアプローチで、そこでは a(==) は ''a と書かれ、「eqtype 変数」と呼ばれている。

多相の等値は、ランタイムシステムの実装者にある種のことを要求する。例えば、Standard ML の参照型は、その他の型とは異なるやり方で比較される必要がある。そのため、ランタイムにおいては、参照と他のポインタは区別できなければならない。

オブジェクト指向プログラミング

多相の等値が、抽象データ型上のユーザ定義による等値演算にまで拡張できると良いだろう。これを実装するには、各オブジェクトが、等値検査を実行する手続きである「メソッド」へのポインタを運んでいなければ

ならない。もし、等値以外にも複数の演算についてこのような性質を持たせようとするなら、各オブジェクトは、適切なメソッドの「辞書」へのポインタを運ぶことになる。これは、まさに、オブジェクト指向プログラミングで使われているアプローチである [GR83]。

このような多相の等値では、等値関数の両方の引数は、いずれも同じ関数へのポインタを持っていることになるだろう（なぜなら、これらはどちらも同じ型であるため)。これは、辞書は引数のオブジェクトとは独立に渡されてもよいことを示唆する。多相の等値には、ひとつの辞書と 2 つのオブジェクト（これらには辞書を含まない）が渡されることになる。これが、本稿で述べられる型クラスと翻訳方法の背後にある洞察である。

3. 初歩的な例

では、ひとつの例を用いて型クラスを導入しよう。

Int, Float の 2 つの型のうえで、(+), (*) および nagate (単項マイナス）を多重定義したいものとする。このために、ひとつの型クラス、num を導入する。この定義は図 1 の class 定義に示される通りである。この定義は次のように読むことができる：「型 a は、(+), (*) および nagate という名の関数が存在して、それらが a の上で定義される適切な型をもつとき、クラス Num に属する」

次に、図 1 の 2 つの instance 宣言にあるように、このクラスのインスタンスを定義していく。Num Int は、「Int の上で定義された適切な型をもつ関数 (+), (*) および nagete が存在する」というような内容を主張している。インスタンス宣言は、これら 3 つの関数について適切な束縛を与えることで、この主張を正当化する。型推論アルゴリズムは、これらの束縛が実際に適切な型を持っていることを検証しなければならない。ここで、適切な型とは、addInt, mulInt にとっては Int->Int->Int 型、negInt にとっては Int->Int 型である。(図 1 の定義では、addInt, mulInt や netInt などが標準プレリュード中で定義されていることを想定している）Num Fload インスタンスも同じように宣言される。

ここで、記法について述べておこう： 型クラス名や型構築子は大文字で始め、型変数名は小文字で始める。ここでは、Num は型クラス、Int と Float は型構築子、そして、a は型変数である。

さて、ここで次の関数を定義する。

```
square x = x * x
```

この定義から square の型を推論するアルゴリズム（これは付録で示される）が存在する。それにより、次のような型が導出される：

```
square :: Num a => a -> a -> a
```

これは、「square は a -> a 型をもつ。ここで、a は Num クラスに属するような任意の型である（すなわち、(+), (*) および nagate は a の上で定義される)」というように読める。ここでは、また、以下のような式を書くことができるようになる。

```
class Num a where
   (+), (*) ::  a -> a -> a
   nagete   ::  a -> a

instance Num Int where
   (+)    =  addInt
   (*)    =  mulInt
   nagate =  negInt

instance Num Float where
   (+)    =  addFloat
   (*)    =  mulFloat
   nagate =  negFloat

square            :: Num a => a -> a
square x          =  x * x

squares           :: Num a, Num b, Num c => (a, b, c) -> (a, b, c)
squares (x, y, z) =  (square x, square y, square z)
```

図 1　算術演算子の定義

```
square 3
square 3.14
```

それぞれについて、適切な型が導出される（前者は Int, 後者は Float)。一方で、square 'x' と書いた場合には、コンパイル時に型エラーとなる。なぜなら、Char は（インスタンス宣言によって）数値型のひとつとは宣言されていないからである。

ここに至って、前述の squares の型を定義すると、図 1 に示される型が推論されるようになった。この型は次のように読める：「a, b, c を　Num クラスに属するような任意の型としたとき、square は、型 (a,b,c) -> (a,b,c) を持つ」（ここで、(a,b,c) は、a, b, c の直積の型である）こうして、squares は、8 つではなく、ただひとつの型を持ち、以下のような項は合法となり、適切な型が導出される。

```
squares (1, 2, 3.14)
```

4. 翻訳

多重定義における本方式の特徴の一つは、クラスとインスタンス宣言を含む任意のプログラムを、これらを用いない等価なプログラムにコンパイル時に変換できる点である。この等価はプログラムは、有効な

```
data  NumD a  =  NumDict (a -> a -> a) (a -> a -> a) (a -> a)

add (NumDict a m n)  =  a
mul (NumDict a m n)  =  m
neg (NumDict a m n)  =  n

numDInt    :: NumD Int
numDInt    =  NumDict addInt mulInt negInt

numDFloat  :: NumD Float
numDFloat  =  NumDict addFloat mulFloat negFloat

square'         :: NumD a -> a -> a
square' numDa x  =  mul numDa x x

squares'  :: (NumD a, NumD b, NumD c) -> (a,b,c) -> (a,b,c)
squares' (numDa, numDb, numDc) (x, y, z)
    =  (square' numDa x, square' numDb y, square' numDc z)
```

図 2　算術演算子の翻訳

Hindley/Milner 型を持つ。

ここでは、その翻訳方法を例を用いて示すことにしよう。図 2 は、図 1 に示された宣言の翻訳である。

各々のクラス宣言に対して、そのクラスの適切な「メソッド辞書」に対応するひとつの型と、その辞書のメソッドにアクセスするための関数群を導入する。この例では、図 2 に示す通り、Num クラスに対応して NumD クラスを導入している。data 宣言は、この新しい型の型構築子として NumD を定義する。この型の値は、値構築子 NumDict を用いて生成され、図 2 中で示されているような型の 3 つの要素を持つ。関数 add, mul および neg は、NumD 型の値を引数としてとり、それぞれ、その第一、第二、および、第三要素を返す。

Num クラスのインスタンスは、いずれも、NumD 型を持つ値の宣言に翻訳される。したがって、Num Int インスタンスに対応して、NumD Int 型のデータ構造を定義し、Float に関しても同様に定義する。

そこで、x+y, x*y および nagate x の各項は、それぞれ次に示すような項に置き換えられる。

```
x+y --> add numD x y
x*y --> mul numD x y
nagate x --> neg numD x
```

ここで numD は適当な辞書である。では、この適当な辞書はどのように求められるだろうか。例えば、以下のような翻訳が得られる：

```
3 * 3 --> mul numDInt 3 3
3.14 * 3.14 --> mul numDFloat 3.14 3.14
```

これらをベータ簡約して、それぞれ、 mulInt 3 3 や mulFloat 3.14 3.14 のように変換し最適化することは、

コンパイラにとって容易である。

関数の型にクラスが含まれる場合、このクラスは、実行時に渡される辞書に変換される。例として、以下の square の定義を型とともに示す：

```
square :: Num a => a -> a
square = x * x
```

これは、以下のように翻訳される：

```
square' :: NumD a -> a -> a
square' numD x = mul numD x x
```

square の各適用は、この追加パラメータを渡すように翻訳されなければならない：

```
square 3 --> square' numDInt 3
square 3.2 --> square' numDFloat 3.2
```

最後に、squares の変換も図 2 に示した。このように、型は 8 つでなく 1 つのみ、翻訳も 8 つではなく、ただ 1 つだけとなっている。指数関数的増大は避けることができた。

5. より進んだ例： 等値

本節では、型クラスとインスタンスの宣言によって等値を定義する方法を述べる。型クラスは、Standard ML で用いられている「等値型変数」を素直に一般化したものとなっている。Standard ML と違う点は、この機構においては、ユーザが抽象データ型にも無理なく等値を拡張できるところである。さらに Standard ML と違う点は、この機構はコンパイル時に翻訳してしまうことが可能であり、ランタイムの実装者になんら特別な要求をしないところである。

定義は図 3 に要約してある。まず、Eq クラスを宣言するところから始める。このクラスはひとつの演算子、(==) をもつ。つぎに、このクラスのインスタンスである Eq Int および Eq Char を宣言する。

さらに、図 3 に示すとおり、通常の方法で member 関数を定義する。この member 関数の型は、型推論が可能であるため、明に与える必要はない。推論される型は、

```
member :: Eq a => [a] -> a -> Bool
```

これは、「member は a を Eq クラスに属する（つまり a における等値が定義されている）ような任意の型としたとき、[a] -> a -> Bool 型をもつ」と読む。（これは、Standard ML における型 "a list -> "a -> bool と

```
class  Eq a  where
  (==) :: a -> a -> bool

instance  Eq Int  where
  (==)  =  eqInt

instance  Eq Char  where
  (==)  =  eqChar

member                  :: Eq a => [a] -> a -> Bool
member [] y             =  False
member (x:xs) y         =  (x == y) \/ member xs y

instance  Eq a, Eq b => Eq (a,b)  where
  (u,v) == (x,y)        =  (u == x) & (v == y)

instance  Eq a => Eq [a]  where
  [] == []              =  True
  [] == y:ys            =  False
  x:xs  ==  []          =  False
  x:xs  ==  y:ys        =  (x == y) & (xs == ys)

data  Set a  =  MkSet [a]

instance  Eq a => Eq (Set a)  where
  MkSet xs == MkSet ys  =  and (map (member xs) ys)
                        & and (map (member ys) xs)
```

図 3　等値の定義

まったく等価である。ここで”a は「等値型変数」を表す。）そうすると、以下のような式を書くことができ、これらはいずれも True と評価される。

```
member [1, 2, 3] 2
member "Haskell" 'k'
```

その次に、ペアに対する等値を定義するインスタンスを示してある。このインスタンスの最初の行は、「任意の型 a, b がいずれも Eq に属するとき、ペア (a, b) も Eq に属する」と読む。言いかえると、「もし、a について等値が定義され、かつ、 b について等値が定義されるならば、(a, b) についての等値が定義される」となる。このインスタンスは、ペアに対する等値を、通常のやり方で、二つの要素の等値で定義する。

同様に、リストの等値を定義することもできる。このインスタンスの最初の行は、「もし a の等値 g 定義されているなら、『a のリスト』の等値が定義される」と読む。ここで、次のような式を書くことができ、これらはいずれも False に評価される。

```
"hello" == "goodbye"
[[1, 2, 3], [4, 5, 6]] == []
```

```
data  EqD a     =  EqDict (a -> a -> Bool)

eq (EqDict e)   =  e

eqDInt          :: EqD Int
eqDInt          =  EqDict eqInt

eqDChar         :: EqD Int
eqDChar         =  EqDict eqChar

member'                          :: EqD a -> [a] -> a -> Bool
member' eqDa [] y                =  False
member' eqDa (x:xs) y            =  eq eqDa x y \/ member' eqDa xs y

eqDPair                          :: (EqD a, EqD b) -> EqD (a,b)
eqDPair (eqDa,eqDb)              =  EqDict (eqPair (eqDa,eqDb))

eqPair                           :: (EqD a, EqD b) -> (a,b) -> (a,b) -> Bool
eqPair (eqDa,eqDb) (x,y) (u,v)   =  eq eqDa x u & eq eqDb y v

eqDList                          :: EqD a -> EqD [a]
eqDList eqDa                     =  EqDict (eqList eqDa)

eqList                           :: EqD a -> [a] -> [a] -> Bool
eqList eqDa [] []                =  True
eqList eqDa [] (y:ys)            =  False
eqList eqDa (x:xs) []            =  False
eqList eqDa (x:xs) (y:ys)        =  eq eqDa x y & eq (eqDList eqDa) xs ys
```

図 4　等値の翻訳

```
member ["Haskell", "Alonzo"] "Moses"
```

最後のデータ宣言は新たな型構築子 Set と、新たな値構築子 MkSet を定義している。もし、モジュールが Set をエクスポートしつつ MkSet を隠すと、モジュールの外からは Set の表現にアクセスできなくなる。これは、Haskell で抽象データ型を定義するときに用いられている機構である。最後のインスタンスは集合に対する等値を定義している。このインスタンスの最初の行は、「a の等値が定義されているなら、『a の集合』の等値が定義される」と読む。ここでは、集合はリストで定義される。二つの集合は、前者のリストの全要素が後者のリストに含まれ、かつ、その反対も成り立つ場合に等しいとされる。（この定義においては、標準関数である map と and が用いられている。map は、ある関数を、あるリストの各要素に適用する。and は、真理値リストの論理積を返す。）集合の等値が member を用いており、さらに、member が多重定義された等値を用いていることから、整数の集合や、整数のリストの集合や、さらには、整数の集合の集合に対しても、等値を適用することができる。

この最後の例は、型クラス機構によって、抽象データ型に対する関数の多重定義が、自然な形で可能となる様を示している。特に、この例は　Standard ML や Miranda で提供される等価を改善するものとなっている。

5.1 等値の翻訳

では、前述した等値の例に対する、翻訳機構の適用について考えよう。図 3 にある宣言の翻訳を図 4 に示す。この翻訳の最初の部分には新しいところはなく、4 節で示した翻訳と似通っている。

最初に、Eq クラスに対応する辞書 EqD を定義する。ここでは、クラスは唯一の演算子 (==) を持ち、よって辞書はただ一つのエントリを持つ。セレクタ関数である eq は、EqD 型のひとつの辞書をとり、その唯一のエントリ（これは a->a->Bool 型をもつ）を返す。インスタンス Eq Int と Eq Char に対応して、EqD Int 型と EqD Char 型の二つの辞書が定義される。これらの辞書は適切な等値関数を持っており、member 関数は素直に member' へと翻訳さえる。以下に、3 つの式とその翻訳の例を示す：

```
3*4 == 12
 --> eq eqDInt (mul numDInt 3 4) 12

member [1, 2, 3] 2
  --> member' eqDInt [1, 2, 3] 2

member "Haskell" 'k'
 --> member' eqDChar "Haskell" 'k'
```

リストに対する等価の宣言を翻訳は、もう少しトリッキーなやり方になる。インスタンス宣言を再掲すると、以下の通り。

```
  instance Eq a => Ea [a] where
  ...
```

これは、もし型 a に等値が定義されているなら [a] 型に対して等値が定義されると述べている。これに対応し、型 [a] に対するインスタンス辞書は、型 a に対する辞書によってパラメータ化され、結果として次のような型となる：

```
  eqDList :: EqD a -> EqD [a]
```

翻訳の残りの部分は図 4 に示してある通り、ペアに対する等値とほとんと同じである。以下に、例として 3 つの式とその翻訳を示す：

```
 "hello" == "goodbye"
   --> eq (eqDList eqDChar) "hello" "goodbye"
```

```
[[1, 2, 3], [4, 5, 6]] == []
  --> eq (eqDList (eqDList eqDInt)) [[1, 2, 3], [4, 5, 6]] []

member ["Haskell", "Alonzo"] "Moses"
  --> member' (eqDList eqDChar) ["Haskell", "Alonzo"] "Moses"
```

最適化のひとつとして、eq (eqDList eqD) をベータ簡約し eqList eqD に変換する（ここで eqD は任意の等値の辞書）のは、コンパイラにとって簡単である。この最適化は、上の最初の 2 つの例、および、図 4 にある eqDList の定義そのものにも適用できる。

ここで示す翻訳と、Standard ML や Miranda の多相等値の効率は比較する価値がある。eqInt などの個々の演算については、翻訳の方が引数が予め判明している分、多相等値よりも少し効率が良い。その一方で、member や eqList においては、明示的に等値演算子を渡す必要があり、これは多相等値にはなかったオーバーヘッドである。これらのトレードオフを評価するには、より詳細な実験が必要である。

6. サブクラス

これまで、Num と Eq を完全に別の型クラスとして考えてきた。なので、もし算術演算子と等値演算子の両方を用いたければ、それらが別々に型の中に出現することになる：

```
memsq :: Eq a, Num a => [a]->a->Bool
memsq xs x = member xs (square x)
```

実際のところ、これは少々奇妙である。我々は、(+), (*) および negate が定義されているようなデータ型には (==) も定義されていることを期待する。ただし、その逆は成り立たない。したがって、Num を Eq のサブクラスにするのが良さそうである。

それは、次のように書くことで可能である：

```
class Eq a => Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
```

これは、a が Eq クラスにも属する場合に限り、Num クラスに属し得ることを表明している。言い換えると、Num は Eq のサブクラスであり、または、同じことだが、Eq は Num のスーパークラスである。これによってインスタンス宣言は変わらないが、Num Int インスタンス宣言が有効になるのは、同一スコープに Eq Int 宣言がある場合のみとなる。

これによって、Num a を含む型はいかなるときにも Eq a も含むことが保証されるので、簡便のために、Num a があるときには、型から Eq a を省略することが可能となる。つまり、先ほど示した memsq の型を表すのに次のように書ける：

```
memsq :: Num a => [a] -> a -> Bool
```

Eq a 修飾子は、Num a によって暗黙のうちに示されるので、もはや言及する必要がない。

一般に、型は任意の個数のサブクラス、ないし、スーパークラスを持つことができる。以下は人為的な例である：

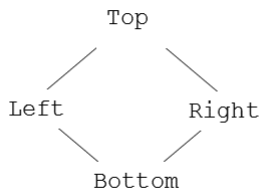
```
class Top a where
  fun1 :: a -> a

class Top a => Left a where
  fun2 :: a -> a

class Top a => Right a where
  fun3 :: a -> a

class Left a, Right a => Bottom a where
  fun4 :: a -> a
```

これらの型同士の関係を図示すると以下のようになる：



複数のスーパークラスは、通常のオブジェクト指向プログラミング言語の実装においては、しばしば問題を引き起こすが、本稿に示した翻訳の枠組みでは問題とならない。翻訳は、単純に、適切な辞書が実行時に渡されることを保証する。ここでは、いくつかのオブジェクト指向実行系で行われているような、特別なハッシュ法は不要である。

7. さいごに

型クラスの宣言に、各インスタンスが満たすべき性質を規定するようなアサーションを加えようと考えるのは、自然なことである。

```
class Eq a where
  (==) :: a -> a -> Bool
  % (==) は同値関係である

class Num a where
  zero, one :: a
  (+), (*) :: a -> a -> a
  negate :: a -> a
  % (zero, one, (+), (*), negate) は環を構成する
```

各インスタンスがこれらの性質を満たすことが証明されている場合に限り、これらの性質に依存した証明が有効となる。ここでは、アサーションは単にコメントとして書かれているのみである。より洗練されたシステムであれば、これらのアサーションを検証するかもしれない。そうなると、型クラス宣言は、OBJ [FGJM85] におけるオブジェクト宣言と似たものとなってくる。このことは、型クラスとオブジェクト指向について、異なる種類の関係性を示唆している。

上述の例における zero, one のように、多重定義された定数をつくることも可能である。しかしながら、定数の無制限な多重定義は、追加の型情報なしには多重定義が解決できないような状況を生じさせる。たとえば、one * one なる式は、これが Int になるのか Float になるのかを特定する文脈なしには意味を持たない。なので、本稿では、3 なら Int 型, 3.14 なら Float 型というように、型が曖昧にならないよう注意して選んだ定数を用いた。より一般的に定数を扱うには、派生型間の型強制が必要と思われる。

型クラスを複数の型変数に適用できるようにしておくのは合理的である。例えば、次のようなもの：

```
class Coerce a b where
  coerce :: a -> b

instance Coerce Int Float where
  coerce = convertIntToFloat
```

この場合、Coerce a b という表明は、a は b の部分型のひとつであるという表明と同じであるととらえられる。これは、本稿の内容と、有界量化、および、派生型の関係を示唆している。（この分野の優れたサーベイとしては、[CW85, Rey85] を参照。また、より新しい仕事として [Wan87, Car88] がある）

型クラスは、ある種の有界量化子であるとみなすことが可能であり、型変数が実体化し得る型の範囲を制限する。しかし、他の有界量化へのアプローチとは違い、型クラスは暗黙の型強制（たとえば、サブクラス Int からスーパークラス Float へ、または、x, y および z フィールドを持つレコードから、x, y を持つレコードへ）を導入しない。型クラスを他のアプローチの関係についてより深く調べることは、きっと面白いだろう。

型クラスは、また、抽象データ型の一種とみなすこともできる。型クラスは、それぞれ、関数のあつまりと、それらの型を規定する。しかし、それらの関数をどのように実装するかについては定めない。ある意味では、

各型クラスは、ある抽象データ型の複数の実装に対応し、１つの実装は各々インスタンス宣言に対応する。型クラスと抽象データ型に関する現代的な研究成果［CW85, MP85, Rey85］の関係についても、ぜひ探究して頂きたい。

Kaes の成果については既に言及した。我々のものを彼の方法と比べたときの、概念的、および、記法的な利点は、多重定義される関数をクラスによってグルーピングできる点である。さらに、我々のシステムの方が、より一般的である。Kaes は、前述の coerce の例のような、二つ以上の型変数に関わる多重定義を扱わなかった。最後に、我々の翻訳規則は、彼の方法を改善したものとなっている。Kaes は、翻訳規則の集合（彼はこれを「意味論」と呼んだ）を２つ示している。ひとつが静的、もう一つが動的である。彼の動的意味論は、本稿で示した言語に比べ、そのパワーの点で制限されている。静的意味論の方は、パワーの点では同等だが、本稿の翻訳規則とは異なり、プログラムサイズの著しい増大を招き得る。

我々の翻訳方式における、ひとつの欠点は、実行時に渡される引数が一つ増えることである。この増えた引数は、メソッド辞書に対応するものである。部分評価［BEJ88］を用いて、特定の辞書にむけに特化したバージョンの関数を生成することによって、この余分な引数のコストをいくらか減じることは可能だろう。これは、コードサイズと引き換えに、実行時間を削減する。我々の方法（部分評価あり、なしともに）と他の技法の間のトレードオフを評価するには、さらなる研究が必要である。

上述の通り、明かに、まだ探究すべき様々な問題が残っており、評価しなければならないトレードオフもたくさんある。Haskell が型クラスを提供するので、これによる実際的な実験に期待している。

謝辞

多重定義は関数の型に反映されるのではないかという重要な洞察は、（少々異なる形で） Joe Fasel によって提案された。さらに、以下に示す Haskell 委員会メンバ、および、IFIP 2.8 メンバの方々との議論やコメントに感謝します：Luca Cardelli, Bob Harper, Paul Hudak, John Hughes, Stefan Kaes, John Launchbury, John Mitchell, Kevin Mitchell, Nick Rothwell, Mads Tofte, David Watt.

付録、参考文献

（あとで訳す[*5]）

[*5] すみません、すみません。１５日に間に合わなかったけど、付録も訳します。