ソフトウェア基礎論　15－4
（１９８５．１２．２０）

論理型言語と関数型言語の関数的統合に関する試み

中島秀之・戸村　哲
（電子技術総合研究所）

1.はじめに

論理型言語と関数型言語の融合に関する研究が盛んである．両者を融合する目標としては，“人間にとって自然な表記”が挙げられる．関数と述語と間の関係は，述語を関数の特殊例として値が Boolean型である関数と考えることもできるし，逆に関数を述語の特殊例として関係が一意に定まる述語と見ることもできるので，両者は記述力では差はないと考えられる．従って，融合は記述力の増加ではなく，表現の自然さを中心に考えるべきである．言語を設計する目標には“使用するのに十分な表現力と実用に耐える効率”を達成することがあるが，この点については別稿に譲る．

論理型言語と関数型言語を融合させる方法としては，全体を関数部分と論理（関係）部分とに分け，両者に別々の意味を与えるアプローチもあるが，ここでは単一の枠組みの下に両方の表記を許すシステムを考える．

項記述［Nakashima 1985，戸村 1985 ］では述語を基本にして論理の意味の下に両者を統合する試みを行なったが，本論では関数から出発し，項書換えによって両者を統合するアプローチを提案する．

2.述語と関数の変換

本節では，関数または書換えを基本として（狭義の）述語を表現したものと（狭義の）関数を表現したものとを自然な意味で同一視するための考え方を示す．

ここでいう関数とは一般の写像であり，特に一価性などは仮定していない．述語は真偽値（Boolean)を値域とする関数として表現する．

例として２個のリストを連結したリストを返す関数 append を考える．関数appendはリスト２個からリストへの関数であるから，その関数としての型を，

append：List×List→List

と書くことにする．ここでListはリスト型，BoolはBoolean 型である．

このとき x,yをそれぞれリスト型とすると，append(x,y) はリスト型である．なお w,x,y,zは変数を表わし，それ以外は定数を表わすものとする．関数をcurry 化すると，appendの第一引数を与えた

append(x)

は，残りのリスト１つからリストへの関数であるから，

append(x) ：List →List

であり，

append(x,y) ≡ append(x) (y)

が成立する．

次に同じ関数をリスト３つの間の述語として場合

append'(x,y,z)

を考える．append' とappendとは

append'(x,y,z)≡ (append(x,y) = z)

という関係になる．

この述語append'を関数として扱うと，３つのリストから Booleanへの関数

append' ：List×List×List→Bool

である．このappend' の curry化を考えてみる．引数を一つだけ与えた

append'(x)

は，残りのリスト２つを受取って Booleanを返す関数であるから，

append'(x)：List×List→Bool

である．同様にして

append'(x,y)：List→Bool

append'(x,y,z)：Bool

である．

さてappend'(x,y)の型はList→Boolであるのに対し， append(x,y)の型はListである．この両者を同じものとして扱うには，List→BoolとListを互いに変換可能とみなす必要がある．

そのために，すべてのデータを Booleanへの関数と考える．即ち，項τが関数ではなくデータである場合に，このτを自分自身の同定関数，つまり一引数関数で引数がτ自身に等しい時にtrueを返し，そうでない場合にはfalse を返す関数とみなす．

今，関数 fが引数 xに対して値として bを返すことを項の書換えを用いて

f(x) ==> b

と表現し，またこの書換え自身をラムダ抽象化を用いて

λ(x)b

と表現することにする．この記法を用いるとデータτの関数としての解釈は部分的には

τ（τ）==> true

と表現できる．例えばa を定数とし，consをデータ構成子とすると

a(a) ==> true

cons(1,2)(cons(1,2)) ==> true

が成立する．言い換えると

a とλ(a) true

cons(1,2) とλ(cons(1,2)) true

をそれぞれ同一視するのである．

データと同定関数とを同一視することで，関数を自然な述語として解釈でき，逆に述語を自然な関数として解釈できること，つまり両者を自然に統合できることを示す．

まずデータ型Tを関数T→Boolとみなすことで，関数を述語として解釈できる．関数appendの型はList×List→Listであるが，このappendに３つのリストを適用した場合を考える．例えば

append(cons(1,nil),cons(2,nil),cons(1,cons(2,nil)))

は curry化すると，

append(cons(1,nil),cons(2,nil)) (cons(1,cons(2,nil)))

となる．関数の部分であるappend(cons(1,nil),cons(2,nil)) を計算すると

cons(1,cons(2,nil))

となるので，全体は

cons(1,cons(2,nil)) (cons(1,cons(2,nil)))

となる．従ってcons(1,cons(2,nil)) を同定関数と解釈すると，値としてtrueを得る．このように関数appendは３つのリストをもらってBoolean を返す関数，

append: List×List×List→Bool

としても解釈でき，

append(x,y)(z)≡append'(x,y,z)

が成立する．つまり関数appendを述語append' と見なすことができる．

次に関数Ｔ→Boolをデータ型Ｔとみなすことで，述語を関数として解釈できる．例えば

append'(x,y)

の型はList→Boolであるが，これをListとみなすと，例えば

append'(append'(x,y),z,w)

と書くことができる．

更にBoolをBool→Boolと見なすことによって述語，即ち Booleanへの関数自身が更に Booleanの値を持つことになる．例えば

append'(x,y,z)：Bool

を

append'(x,y,z)：Bool→Bool

とみなすと，これは

append' ：List×List×List×Bool→Bool

を curry化したものと考えることもできる．

また Booleanの定数true,falseは定義によると

true(true) ==> true

true(false) ==> false

false(true) ==> false

false(false) ==> false

が成立するので，

append'(cons(1,nil),cons(2,nil),cons(1,cons(2,nil)),true) --> true

append'(cons(1,nil),cons(2,nil),cons(1,cons(2,nil)),false) --> false

append'(cons(1,nil),cons(2,nil),cons(1,cons(2,nil)),false､false) -->true

append'(cons(1,nil),cons(2,nil),cons(3,nil),false) --> true

などが成立する．

一般には，任意個数のtrue，false の列を関数の最後につけてもよく，全体がそれに応じた真偽値を持つことになる．

3.関数型言語 uni

uni は前節で述べた考え方に基づき，述語を関数の上に自然に統合した関数型言語である．本節ではuniについて具体的に述べる．

[uni オブジェクト]

uni ではオブジェクトはデータと関数の２種類からなる．名前には，関数名，データ構成子名，変数名がある．書換え規則の与えられた名前 fは，対応する関数を表わす．それ以外の名前 dはデータ構成子であり，

名前 dに引数を付けた d(a1,...an)はデータを表わす．変数はその値であるオブジェクトを表わす．

[関数オブジェクトの表現]

書換え規則は関数の定義域と値域との対応関係の一部を与える．関数は書換えの集合として定義される．例えば，Boolean 上の関数 notはtrueに対してはfalse を，false に対しては true を値として取る．これを書換え規則で表現すると，

not(true) ==> false

not(false) ==> true

となる．この２つの書換え規則そのものはラムダ式を用いるとそれぞれ，

λ(true) false

λ(false) true

となる．not の表わす関数そのものはこれら対応関係の集合であるが，そのことを表現するのにΠ記号を用いて

Π (λ(true) false ; λ(false) true)

と書く．この表記を用いると関数 notは

not ≡Π (λ(true) false ; λ(false) true)

と表現できる．

[関数の定義形式]

uni では関数を書換え規則を用いて定義する．

例に用いたappendは次の２つの書換え規則で定義できる．

まずリストを返す関数appendf として定義するには，

appendf(nil,x) ==> x

appendf(cons(x,y),z) ==> cons(x,appendf(y,z))

とし，また述語として定義するには，

appendp(nil, x, x) ==> true

appendp(cons(x,y),z,cons(x,w)) ==> appendp(y,z,w)

とすればよい．しかしながらuni ではappendf やappendp のどちらの形式でappendを定義しても，これらは関数と述語のいづれとしても全く同じ働きをする．この意味で両者は等価である．つまり，関数あるいは述語のどちらか一方として定義すれば，それを両方の形で使うことができる．利用者は場合に応じて自然な表現方法を選択することができる．従って以下ではappendf とappendp のような関数と述語の区別は行なわない．

このように同じ関数記号が引数の個数によって述語を表現したり，関数を表現したりできる．そこで関数の型を表現するのにその引数の個数と組にして，

append/3 :List×List×List→Bool

append/2 : List×List→List

のように指定することにする．

また uniでは関数を関数記号から関数オブジェクトへの書換えとして定義することもできる．関数appendは関数記号appendからラムダ式への書換え

append ==> λ(nil, x, x) true

append ==> λ(cons(x,y),z,cons(x,w)) append(y,z,w)

と書くこともできる．これはΠ式を用いると

```
append ==>  Π (λ(nil, x, x) true ;
                λ(cons(x,y),z,cons(x,w)) append(y,z,w) )
```

となる．この場合，このほかにappendの書換え規則を定義してはならない．

さてここで注意しておく必要があるのは，uni では関数を一価関数に制限していない点である．また関数の定義を“書換え規則”という形式で与えるが，これは書換え規則の左辺と右辺が等しいことを意味するものではない．従って書換え規則は合流性を満たす必要はない．書換え規則が合流性を満足していない場合 uniではそれは多価関数を定義していると解釈する．例えば，関数 fを

```
f ==> 1
f ==> 2
```

のように定義できる．また

```
f(g(x)) ==> h(x)
g(a)==> b
```

と関数を定義した場合にf(g(a)) は次の２通りの値を持つと解釈する．

```
f(g(a)) --> f(b)
f(g(a)) --> h(a)
```

このように解釈する理由は，uni では項の変形にnarrowing [Hullot 1980] を用いるからである．項の変換で，項の書換え規則の左辺と項とを照合するのに，一方向の照合(match) を用いるのを書換え(rewriting) といい，両方向の単一化(unify) を用いるをnarrowing と呼ぶ．いずれの場合も，照合が成功した場合のに変形が行なわれる．書換えでは変形する項には変数は含まれず，基底項だけが対象となる．これに対し，narrowing は対象とする項に含まれる変数に色々な基底項の置き換えをしたものすべての書換えを実行することに相当する．このため仮に基底項に対する書換え規則が停止性や合流性を満足し，項の表現に一意表現（正規形）が存在する場合であっても，narrowing による結果は一意にならない．従って，変形の一意性を保つような制限をシステムに加えても意味がないと考えるからである．さらに項の間の同値関係や関数の一意性を定義するには，書換えとは別に定義する方が自然であると考えている．

[計算機構]

uni の計算機構は変形すべき項に対して，書換え規則（定義されたものと，データを同定関数と解釈したもの）を適用してnarrowing (>--->) を行なう．narrowing できない既約形が得られるまでこれを繰り返す．一般にuni では既約形は一意ではないので，利用者から他の既約形を探索する指令を与えることができる．

例を示す．下線が付いているのが，システムの出力である．

```
uni> f(a) ==> h(a)
uni> f(b) ==> g(b)
uni>  f(x)
f(a) >---> h(a)  ;
f(b) >---> g(b)
uni>
```

uni で用いるnarrowing は，書換え規則 l ==> rの左辺 lとnarrowing をする項 tとを照合する単一化 unify(l,t) を拡張している．これはデータ tとその同定関数λ(t)、trueとを同一視するためである．基底

データと関数とを単一化するには，関数にデータを適用した値がtrueになるとき照合が成功し，それ以外失敗することになる．

4.例

例 1：

国の人口密度をその国の人口と面積から計算するには，

```
population-density(country) ==> div(population(country),area(country))
```

と定義できる．ここで関数の型は，

```
population/2：Countory×Number→Bool
population/1：Country →Number
area/2：Country ×Number→Bool
area/1：Country →Number
```

である．また

```
div/2 ：Number×Number→Number
```

であるから，

```
population-density/1：Country →Number
population-density/2：Country ×Number→Bool
```

である．

例 2：

条件式if-then-elseは以下のように定義できる：

```
if-then-else(true, x, y)＝＞x
if-then-else(false, x, y) ＝＞y
```

if-then-elseの条件部を扱うにはuni 上に論理演算を定義する必要がある．論理演算を定義するのに，否定(false) の扱いに関しては閉世界仮説：“trueに書換えられないものはすべてfalse とする”の立場をとることにする．ここではシステムで

```
false(x) ==> x=/=true
```

に相当するような書換え規則を導入したものと仮定して話を進める．論理式に関する書換え規則は，trueに関するものだけを与えればよく，

```
and(true,y) ==> y
or(x,true) ==> true
or(true,y) ==> true
```

のように書ける．これらは論理結合子の演算規則である．変数は，すべて式の先頭で全称限量されているものとし，存在限量に関しては関数記号を用いて表わすことにする．この他に公理を追加するためのassertが必要である．assertは次の書換え規則を追加する．

```
assert(imply(x,y))：   y ==> x
```

即ち，y の真偽値はx の真偽値に帰着できるという意味である．この場合，y ==> x は論理的に等価な書換えではないので，x がfalse になったからと言ってy がfalse とはいえず，別の書換え規則を試みる必要がある．uni での書換えはそのように定義されている．即ち，ある項（ここでは論理式）の別の項（ここでは

true）への書換えが失敗するのはすべての可能性を試みた後である．

5．まとめ

関数を基本としてデータをそれ自身の同定関数と同一視することで，述語表現と関数表現を統合するシステムuni を提案した．uni では関数として定義したものを述語のように呼び出すこともできるし，その逆に述語として定義したものを関数として呼び出すこともできる．また，述語の真偽をさらに引数として取る高次の述語も実現できる．

参考文献

Hullot, Jean-Marie:" Canonical Forms and Unification", Lecture Notes in Computer Science, Vol.87 (1980), pp. 318-334

Nakashima, H. :"Term Description: A Simple Powerful Extension to Prolog Data Structures," Proc. of IJCAI IX(1985) ,pp.708-710

戸村　哲：“TDProlog: 項記述可能なProlog処理系”，Proc. of the Logic Programming Conference '85 (1985) .