Type Systems and Programming Languages

前田俊行

平成 13 年 7 月 2 日

20 Recursive Types

Lisp のリストを扱うことを考えてみる。

11 章では言語を拡張することでリストを実現した。しかし、 リストやツリーなどを実現するために場当たり的に言語を拡張するのはうまくない。

そこで、これらの構造を定義するための仕組みが必要となる。Recursive Type がまさにその仕組みである。

ここで、整数のリストをあらわす型 NatList を定義する方法について考えてみる。NatList の値は次の 2 種類であらわせる。

- 空リスト nil
- Nat と NatList のペア

11 章で定義した variant 型と tuple 型を用いるとこのことは次のようにあらわせる。

$$\mathsf{NatList} = \langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\ldots, \ldots\} \rangle\,;$$

タグ nil はそれ自体でリストが空であることがわかるので、上のように単に Unit 型を用いる。

一方タグ cons は Nat と NatList のペアであるので、以下のようにペアの第 1 要素の型は Nat となる。

$$\mathsf{NatList} = \langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat}, \ldots\} \rangle\,;$$

そして、ペアの第 2 要素は NatList 、つまり、今まさに定義しようとしているものであるので以下のようになる。

$$\mathsf{NatList} = \langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat}, \mathsf{NatList}\} \rangle\,;$$

上の定義を見ると、定義しようとしている型 NatList が、その定義中に再帰的にあらわれている。このように再帰的に定義される型を Recursive Type と呼ぶ。

以降では、Recursive Type を名前をつけることなく扱うことができるように、次のような書き方をすることにする。

$$\mathsf{NatList} = \mu \mathsf{X}.\, \langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat}, \mathsf{X}\} \rangle\,;$$

上の定義は「型 NatList を、方程式 $\mathsf{X} = \langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat}, \mathsf{X}\} \rangle$ を満たすような型とする」という意味である。

20.1 Examples

Lists

上の NatList の定義をもとに、NatList の例をさらに続ける。

まず、空のリストをあらわす nil は次のように定義できる。

$$\begin{array}{lll} & \mathsf{nil} & = \quad \langle \mathsf{nil} = \mathsf{unit} \rangle \ \text{as NatList;} \\ \triangleright & \mathsf{nil} & : \quad \mathsf{NatList} \end{array}$$

また、Nat を NatList の先頭に挿入して、新しい NatList を返す関数 cons は次のように定義できる。

$$\begin{array}{lll} & \mathsf{cons} & = \quad \lambda \mathsf{n} : \mathsf{Nat}.\ \lambda \mathsf{l} : \mathsf{NatList}.\ \langle \mathsf{cons} = \{\mathsf{n}, \mathsf{l}\} \rangle \ \text{as NatList;} \\ \triangleright & \mathsf{cons} & : \quad \mathsf{Nat} \to \mathsf{NatList} \to \mathsf{NatList} \end{array}$$

さらに、リストが空であるかどうかを判定する関数 isnil は次のように定義できる。

$$\begin{array}{lll} & \mathsf{isnil} & = \quad \lambda \mathsf{l} : \mathsf{NatList}.\ \mathsf{case}\ \mathsf{l}\ \mathsf{of} \\ & & \qquad\qquad \langle \mathsf{nil} = \mathsf{u} \rangle \Rightarrow \mathsf{true} \\ & & \qquad \mid \quad \langle \mathsf{cons} = \mathsf{p} \rangle \Rightarrow \mathsf{false} \\ \triangleright & \mathsf{isnil} & : \quad \mathsf{NatList} \to \mathsf{Bool} \end{array}$$

また、リストの先頭の整数値を返す関数 hd は次のように定義できる。

$$\begin{array}{lll} & \mathsf{hd} & = \quad \lambda \mathsf{l} : \mathsf{NatList}.\ \mathsf{case\ of}\ \langle \mathsf{nil} = \mathsf{u} \rangle \Rightarrow 0 \mid \langle \mathsf{cons} = \mathsf{p} \rangle \Rightarrow \mathsf{p}.1 \\ \triangleright & \mathsf{hd} & : \quad \mathsf{NatList} \to \mathsf{Nat} \end{array}$$

リストの先頭の要素を除いたリストを返す関数 tl は次のように定義できる。

$$\begin{array}{lll} & \mathsf{tl} & = \quad \lambda \mathsf{l} : \mathsf{NatList}.\ \mathsf{case\ of}\ \langle \mathsf{nil} = \mathsf{u} \rangle \Rightarrow \mathsf{l} \mid \langle \mathsf{cons} = \mathsf{p} \rangle \Rightarrow \mathsf{p}.2 \\ \triangleright & \mathsf{tl} & : \quad \mathsf{NatList} \to \mathsf{NatList} \end{array}$$

使用例

与えられた NatList の全ての値の合計を返す関数 sumlist は、fix を用いて次のように再帰的に定義できる。

```
    sumlist   =   fix (λs : NatList → Nat. λl : NatList.
                    if isnil l then 0 else plus (hd 1) (s (tl l)));
▷   sumlist   :   NatList → Nat
```

sumlist の使用例は次のとおりである。

```
    mylist = cons 2 (cons 3 (cons 5 nil));
    sumlist mylist
▷   10 : Nat
```

Streams

上で考えた NatList は、型としてはリストの長さは無限と考えることができるが、実際の値のリストは、長さは有限である。

そこで次に、実際に無限に要素が存在するような値、Stream を定義する。Stream の型の定義は次のようになる。

```
Stream = μA. Unit → {Nat, A};
```

NatList との違いは、Unit で一段階抽象化されている点である。この抽象化のおかげで、無限に要素を生成することができる。

次に NatList と同じように hd と tl を定義する。

```
    hd   =   λs : Stream. (s unit).1;
▷   hd   :   Stream → Nat
    tl   =   λs : Stream. (s unit).2;
▷   tl   :   Stream → Stream
```

使用例

以下は、0 から順番に整数値 (Nat) を無限に生成する Stream の定義である。fix を用いて再帰的に定義している。

```
    upfrom0   =   fix (λf: Nat→Stream. λn:Nat. λ_:Unit. {n, f (succ n)}) 0;
▷   upfrom0   :   Stream
```

次に hd や tl を使って Stream を操作する。

```
    hd upfrom0
▷   0 : Nat
    hd (tl (tl (tl upfrom0)));
▷   3 : Nat
```

Objects

Recursive Type を用いると、19 章で述べたような purely functional なオブジェクトを定義することができる。例えば、オブジェクト型 Counter は次のように定義できる。

```
Counter = μC. {get:Nat, inc:Unit → C, dec:Unit→ C};
```

この型のオブジェクトを生成するには、fix を用いて次のようにする。

```
c   =  let create = fix (λf: {x:Nat}→Counter. λs: {x:Nat}.
         {get  =  s.x,
          inc  =  λ_:Unit. f {x=succ(s.x)},
          dec  =  λ_:Unit. f {x=pred(s.x)} })
       in create {x=0};
▷  c   :  Counter
```

Counter のメソッドを呼び出すには次のようにする。

```
c1 = c.inc unit;
c2 = c1.inc unit;
c2.get;
▷  2 : Nat
```

Recursive Values from Recursive Types

Recursive Type を用いると、fixed-point combinator を型づけすることができる。例えば、ある型 T に対する fixed-point combinator は次のようにしてつくれる。

```
fix_T   =  λf:T→T. (λx:(μA.A→T). f (x x)) (λx:(μA.A→T). f (x x));
▷  fix_T   :  (T→T) → T
```

上の fixed-point combinator は、型の注釈を除けば untyped-lambda calculus での fixed-point combinator と同じである。

20.2 Formalities

Recursive Type をフォーマルに扱う方法は、例をあげていえば、Recursive Type NatList とその展開形 $\langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat},\ \mathsf{NatList}\}\rangle$ をどのように扱うかで 2 種類存在する。

equi-recursive approach

このアプローチでは、NatList と $\langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat},\ \mathsf{NatList}\}\rangle$ を同じ

型として扱う。つまり、型チェッカにこれらの型が等しいことを自動的に判断させる。ちなみに、この章でここまで述べてきた内容は全てこのアプローチにもとづいている。

このアプローチの利点は、今まで考えてきた言語や型システム、またそれに関する定理やその証明を変更しなくてよい点である。

ただし、型チェッカを変更しなければならない。型チェッカは、Recursive Type(NatList) とその展開された型 ($\langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat},\ \mathsf{NatList}\}\rangle$) が等しいことを判断しなければならない。ところが型チェックアルゴリズムは無限の構造を直接扱うことができないため、それに対する工夫が必要となるのである。このことについては 21 章で述べられる。

また、この方法は後の章で述べる bounded quantification や type operator と組み合わせると重大な理論的困難が生じる。最悪の場合は型チェックアルゴリズムが undecidable になる。

iso-recursive approach

この方法では、NatList と $\langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat},\ \mathsf{NatList}\}\rangle$ を異なる物として扱う。ただし、言語に fold と unfold というプリミティブを用意して、プログラマが明示的に Recursive Type を展開したり、その逆を行ったりできるようにする。

unfold は型の再帰構造を一段階 unfold するものである。例えば、NatList は

$$\mu\mathsf{X}.\,\langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat},\ \mathsf{X}\}\rangle\,;$$

であるが、これを unfold で一段階 unfold すると次のようになる。

$$\langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat},\ \mu\mathsf{X}.\,\langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat},\ \mathsf{X}\}\rangle\}\rangle$$

fold は unfold の逆の作用、つまり型の再帰構造を一段階 fold するものである。例えば、次の型を考える。

$$\langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat},\ \mu\mathsf{X}.\,\langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat},\ \mathsf{X}\}\rangle\}\rangle$$

この型を fold で一段階 fold すると型 NatList が得られる。

$$\mu\mathsf{X}.\,\langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat},\ \mathsf{X}\}\rangle\,;$$

fold と unfold をとりいれた言語は別紙の表 20-1 のとおりである。具体的には fold と unfold は以下のように型を変更する identity 関数である。

$$\begin{array}{lcl} \mathsf{unfold}[\mu\mathsf{X}.\mathsf{T}] & : & \mu\mathsf{X}.\mathsf{T} \to [\mathsf{X}\mapsto\mu\mathsf{X}.\mathsf{T}]\mathsf{T} \\ \mathsf{fold}[\mu\mathsf{X}.\mathsf{T}] & : & [\mathsf{X}\mapsto\mu\mathsf{X}.\mathsf{T}]\mathsf{T} \to \mu\mathsf{X}.\mathsf{T} \end{array}$$

次に、iso-recursive approach の例を見てみる。その前に、便宜のため次の

型 NLBody を定義する。(equi-recursive approach では NatList = NLBody である。)

$$\mathsf{NLBody} \;=\; \langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat},\ \mathsf{NatList}\}\rangle;$$

この NLBody を用いると、NatList のコンストラクタは次のように定義できる。

```
nil  =  fold [NatList] (⟨nil = unit⟩ as NLBody);
cons  =  λn : Nat. λl : NatList. fold [NatList] ⟨cons = {n, l}⟩ as NLBody;
```

また NatList に対する操作は次のように定義できる。

```
isnil   =   λl : NatList.
              case   unfold [NatList] l of
                       ⟨nil = u⟩  ⇒  true
                     | ⟨cons = p⟩  ⇒  false
hd   =   λl : NatList.
             case   unfold [NatList] l of
                      ⟨nil = u⟩  ⇒  0
                    | ⟨cons = p⟩  ⇒  p.1
tl   =   λl : NatList.
             case   unfold [NatList] l of
                      ⟨nil = u⟩  ⇒  l
                    | ⟨cons = p⟩  ⇒  p.2
```

上の例のように、iso-recursive approach ではプログラム中に fold と unfold を書かなければならず、一見面倒なように思える。しかし実際の言語では fold や unfold を言語の他の機構に埋め込むことで直接 fold や unfold を書かずにすむようにしている。例えば、ML では datatype の値を生成するときに fold を呼び、パターンマッチ時に unfold を呼ぶようにしている。つまり、上の NatList の例と同じ場所で fold と unfold が暗黙的に呼ばれる。

20.3 Subtyping

問題

μX.Nat→(Even × X) と μX.Even→(Nat × X) はどのような subtype 関係にあるか?

答え

μX.Nat→(Even × X) <: μX.Even→(Nat × X)

直感的な説明は別紙の図を参照。より詳しい議論は 21 章で行われる。