型輪講資料第20章
Recurisve Types

36118
立沢秀晃*

平成15年6月10日

1 初めに

11章において、リスト構造をプリミティブとして型に拡張をした。しかし、このようにデータ構造ごとにプリミティブを定義しても意味がない。そこで、単純な要素から様々な構造が定義できる一般的な機構として**再帰型**(*recursive types*)を考える。つまり、リスト構造[1]は以下のように考えられるということである。

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

この式は両辺にNatListを含むのでNatListの定義にはなっていない。ここで、このような無限のループ構造を表す記号としてμを導入する。この例として、NatListは以下のように表される。

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

この直感的な意味は「NatListは"$\mathsf{X} = \langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \mathsf{Nat}, \mathsf{X}\rangle$" という方程式を満たす無限型である」ということである。

2 Formalities

再帰型の定式化との方法は2つの方法がある。その二つの本質的な違いは「μX.TとTの関係はどうであるか」という質問[2]によって考えることができる。

2.1 equi-recursive approach

equi-recursive approachでは、先ほどの二つの表現は**定義として同じ**であると扱う。つまり、ある型の項が違う型の項を期待する関数の引数として許されることを保証するのは型チェッカーの役割であるとするわけである。

*e-mail: hideaki@yl.is.s.u-tokyo.ac.jp

[1] この章では自然数型のリストのみを扱う。一般の型に対するリストを議論するためには型操作 (type operators) の機構が必要で、29章で導入される。

[2] より具体的には、「NatList と $\langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \mathsf{Nat}, \mathsf{NatList}\rangle$ の関係はどうか」ということである。

この方法の利点としては今まで議論してきたシステムとは型の表現が再帰的にできるということのみが違うことである。そのため、型表現の帰納法によらないような既存の定義、安全定理 (safety theorem) や証明はそのまま用いることができる[3]。

この方法では、型チェッカーに大きな役割を任せており、もう一つのアプローチである iso-recursive approach の fold や unfold が起こるべき場所を推測する必要がある。しかも、より進んだ型付けの特徴と合わせて考えたときに非常に複雑になり、理論上の重大な困難となったり、決定不能な型チェック問題にさえつながることもある。

2.2 iso-recursive approach

iso-recursive approach では、先ほどの二つの表現は異なるが、**同型**であるとして扱う。形式的には、μX.T の展開は、本体である T の中に現れる全ての X を μX.T で置き換えたもの ([X $\mapsto$ μX.T]T) になる。例えば、

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

は

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

に展開できると言うことである。このシステムでは全ての再帰型 μX.T に対して以下の二つの関数を導入する。

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

これらは二つの同型である型を行き来する関数であり、言語のプリミティブとして用意される (Figure20-1) この二つの関数が同型を形成するということは E-UNFLDFLD の評価規則によって表現されている。つまり、対応した unfold に遭遇した場合に fold を消すわけである[4]。

この方法は再帰型を使うときに fold と unfold を明示しなければならず、記法としては比較的難しい。しかし、実際には他の記法に混ざってしまい、これらの記述は「隠す」ことができる。例えば ML では全ての datatype 定義は再帰型の導入を暗示している。その型の値を作るためにコンストラクタを使うときには全て fold が行われ、パターンマッチに現れる全てのコンストラクタは unfold を行う。同様に、Java では全てのクラス定義が再帰型の導入を暗示しており、オブジェクト内のメソッド呼び出しが unfold を行っている。このようにすることによって、iso-recursive 法が実用的になっている。この例としては、以下のような記述が挙げられる。

ML での記述例

```
type nlbody =
    Nil of unit
  | Cons of int * nlbody

let isnil l = match l with
```

[3] この方法の実装には新たな型チェックアルゴリズムを構成することが必要である。これについては 21 章で扱う。

[4] ここで、fold と unfold に対する型の注釈が違っても良いのは、そのような制限を保証するために実行時には型チェックをするからである。ただし、well-typed なプログラムの評価であれば、この評価規則を適用するときには二つの型は同じになっている

```
    Nil(_) -> true
  | Cons(_) -> false

let hd l = match l with
    Nil(_) -> 0
  | Cons(i1, _) -> i1

let tl l = match l with
    Nil(_) -> l
  | Cons(_, tl) -> tl
```

実際の意味

$$
\begin{aligned}
\mathsf{NLBody} &= \langle \mathsf{nil} : \mathsf{Unit},\ \mathsf{cons} : \{\mathsf{Nat}, \mathsf{NatList}\}\rangle \\
\mathsf{nil} &= \mathsf{fold}[\mathsf{NatList}](\langle \mathsf{nil} = \mathsf{unit}\rangle \mathsf{as\ NLBody}) \\
\mathsf{cons} &= \lambda \mathsf{n} : \mathsf{Nat}.\lambda \mathsf{l} : \mathsf{NatList}.\ \mathsf{fold}[\mathsf{NatList}]\langle \mathsf{cons} = \{\mathsf{n}, \mathsf{l}\}\rangle \mathsf{as\ NLBody} \\
\mathsf{isnil} &= \lambda \mathsf{l} : \mathsf{NatList}. \\
&\quad \mathsf{case\ unfold}\ [\mathsf{NatList}]\mathsf{l}\ \mathsf{of} \\
&\quad \langle \mathsf{nil} = \mathsf{u}\rangle \Rightarrow \mathsf{true} \\
&\quad \langle \mathsf{cons} = \mathsf{p}\rangle \Rightarrow \mathsf{false} \\
\mathsf{hd} &= \lambda \mathsf{l} : \mathsf{NatList}. \\
&\quad \mathsf{case\ unfold}\ [\mathsf{NatList}]\mathsf{l}\ \mathsf{of} \\
&\quad \langle \mathsf{nil} = \mathsf{u}\rangle \Rightarrow 0 \\
&\quad \langle \mathsf{cons} = \mathsf{p}\rangle \Rightarrow \mathsf{p.1} \\
\mathsf{tl} &= \lambda \mathsf{l} : \mathsf{NatList}. \\
&\quad \mathsf{case\ unfold}\ [\mathsf{NatList}]\mathsf{l}\ \mathsf{of} \\
&\quad \langle \mathsf{nil} = \mathsf{u}\rangle \Rightarrow \mathsf{l} \\
&\quad \langle \mathsf{cons} = \mathsf{p}\rangle \Rightarrow \mathsf{p.2}
\end{aligned}
$$

3 Examples

再帰型を使って何ができるかを例を使って示す。

3.1 Lists

先ほど紹介したMLの記述からリスト表現ができるのは明らかであろう。使用例としては、引数に与えられたリストの全ての要素の和を返す関数sumlistである。

$$\mathsf{sumlist} = \mathsf{fix}\ (\lambda \mathsf{s} : \mathsf{NatList} \to \mathsf{Nat}.\ \lambda \mathsf{l} : \mathsf{NatList}.\mathsf{if\ isnil\ l\ then\ 0\ else\ (hd\ l) + (s\ (tl\ l)))}$$

ちなみにこれはMLでは

```
let rec sumlist l = if isnil l then 0 else (hd l) + (sumlist (tl l))
```

と表せる。

3.2 Hungry Functions

再帰型を使うといくつでも引数をとれ、新しくいくつでも引数をとれるような関数を返す関数 (hungry function) を定義できる。

$$\mathsf{Hungry} = \mu\mathsf{A}.\ \mathsf{Nat} \to \mathsf{A}$$

この型の要素は不動点演算子を用いて定義する[5]。

$$\mathsf{f} = \mathsf{fix}\ (\lambda \mathsf{f} : \mathsf{Nat} \to \mathsf{Hungry}.\ \lambda \mathsf{n} : \mathsf{Nat}.\ \mathsf{f})$$

ML では、

```
type hungry =
    int -> hungry

let rec f (n:int) = f
```

と書ける。

3.3 Streams

より、意味のある再帰型としてはストリーム型が定義できる。これは、任意の unit 型の値を引数に取り、そのたびに、自然数と新しいストリームのペアを返すものである。

$$\mathsf{Stream} = \mu\mathsf{A}.\ \mathsf{Unit} \to \{\mathsf{Nat}, \mathsf{A}\}$$

ストリームに対して二つのデストラクタが定義できる。すなわち、ストリームを受け取って、一度 unit を適用したの後のペアの初めの要素を返すものとあとの要素を返すものである。

$$\begin{aligned} \mathsf{hd} &= \lambda \mathsf{s} : \mathsf{Stream}.\ (\mathsf{s}\ \mathsf{unit}).1 \\ \mathsf{tl} &= \lambda \mathsf{s} : \mathsf{Stream}.\ (\mathsf{s}\ \mathsf{unit}).2 \end{aligned}$$

ストリームを作るときには先ほどと同様に不動点演算子を用いる。

$$\mathsf{upfrom0} = \mathsf{fix}\ (\lambda \mathsf{f} : \mathsf{Nat} \to \mathsf{Stream}.\ \lambda \mathsf{n} : \mathsf{Nat}.\ \lambda\_ : \mathsf{Unit}.\ \{\mathsf{n}, \mathsf{f}\ (\mathsf{succ}\ \mathsf{n})\})0$$

これは 0 から昇順の自然数を生成するストリームである[6]。
ML では、

```
type stream =
    unit -> (int * stream)

let upfrom0 =
  let rec f n () = (n, f (n+1)) in f 0
```

[5] つまり再帰的な定義である。
[6] $\mathsf{hd}\ \mathsf{upfrom0} = 0$、$\mathsf{hd}\ (\mathsf{tl}\ (\mathsf{tl}\ \mathsf{upfrom0})) = 2$ ということ。

となる。
ストリームは、自然数を受け取り自然数と新しいプロセスのペアを返すような型であるプロセスとして一般化できる。

$$\mathsf{Process} = \mu\mathsf{A}.\ \mathsf{Nat} \to \{\mathsf{Nat}, \mathsf{A}\}$$

ストリームにたいしてと同様にプロセスに関する補助関数は以下のように定義できる。

$$\mathsf{curr} = \lambda \mathsf{s} : \mathsf{Process}.\ (\mathsf{s}\ 0).1$$
$$\mathsf{send} = \lambda \mathsf{n} : \mathsf{Nat}.\ \lambda \mathsf{s} : \mathsf{Process}.\ (\mathsf{s}\ \mathsf{n}).2$$

プロセスの例としては今まで与えられた自然数の和を計算するプロセスが挙げられる。

$$\mathsf{p} = \mathsf{fix}(\lambda \mathsf{f} : \mathsf{Nat} \to \mathsf{Process}.\ \lambda \mathsf{acc} : \mathsf{Nat}.\ \lambda \mathsf{n} : \mathsf{Nat}.\ \{\mathsf{acc} + \mathsf{n}, \mathsf{f}\ (\mathsf{acc} + \mathsf{n})\})0$$

ML では以下の通り。

```
type process =
    int -> (int * process)

let p =
  let rec f acc n = ((acc+n), f (acc+n)) in f 0
```

3.4 Objects

レコード型を考えるとオブジェクトが表現できる。先ほどは関数を直接用いてペアを返してプロセスを表現したが、レコード型を考えることによって複数の操作を埋め込むことが可能になったからである[7]。例としては以下のカウンタ型が挙げられる。

$$\mathsf{Counter} = \mu\mathsf{C}.\ \{\mathsf{get} : \mathsf{Nat},\ \mathsf{inc} : \mathsf{Unit} \to \mathsf{C},\ \ \mathsf{dec} : \mathsf{Unit} \to \mathsf{C}\}$$

この使用はまた不動点演算子を使う。

$$\begin{aligned}
\mathsf{c} = \mathsf{let\ create} \ = \ & \mathsf{fix}(\lambda \mathsf{f} : \{\mathsf{x} : \mathsf{Nat}\} \to \mathsf{Counter}.\ \lambda \mathsf{s} : \{\mathsf{x} : \mathsf{Nat}\}. \\
& \{\mathsf{get} \ = \mathsf{s.x}, \\
& \mathsf{inc} \ = \lambda_ : \mathsf{Unit}.\ \mathsf{f}\ \{\mathsf{x} = \mathsf{succ}(\mathsf{s.x})\}, \\
& \mathsf{dec} \ = \lambda_ : \mathsf{Unit}.\ \mathsf{f}\ \{\mathsf{x} = \mathsf{pred}(\mathsf{s.x})\}\}) \\
\mathsf{in\ create}\ \{\mathsf{x} = 0\}
\end{aligned}$$

これに対して、以下のように使う。

$$\begin{aligned}
\mathsf{c1} \ &= \ \mathsf{c.inc\ unit} \\
\mathsf{c2} \ &= \ \mathsf{c1.inc\ unit} \\
&\mathsf{c2.get} \\
&\to 2
\end{aligned}$$

ML では以下の通り。

[7] プロセス型も、返すタプルの要素を増やせばこれと同等のことが可能となる。

```
type counter =
    {get:int; inc:unit->counter; dec:unit->counter}

type xrec = {x:int}

let c =
  let rec f s =
    {get = s.x;
     inc = (fun () -> (f {x=(s.x + 1)}));
     dec = (fun () -> (f {x=(s.x - 1)}))} in
    f {x=0}

# let c1 = c.inc ();;
val c1 : counter = {get = 1; inc = <fun>; dec = <fun>}
# let c2 = c1.inc();;
val c2 : counter = {get = 2; inc = <fun>; dec = <fun>}
```

3.5 Recursive Values from Recursive Types

再帰型の表現力の強さを見るため、再帰型を使って不動点演算子を定義してみる。

$$\mathsf{fix}_T = \lambda \mathsf{f} : \mathsf{T} \to \mathsf{T}.\ (\lambda \mathsf{x} : (\mu \mathsf{A}.\mathsf{A} \to \mathsf{T}).\ \mathsf{f}\ (\mathsf{x}\ \mathsf{x}))\ (\lambda \mathsf{x} : (\mu \mathsf{A}.\mathsf{A} \to \mathsf{T}).\ \mathsf{f}\ (\mathsf{x}\ \mathsf{x}))$$

これはちなみに、型を除けば型無しの不動点演算子と同じになる。この定義のポイントは xx という表現である。つまり x は関数型であり、かつその引数の型でなければいけないのだが、それは再帰型の定義 $(\mu \mathsf{A}.\mathsf{A} \to \mathsf{T})$ にちょうど当てはまるわけである。

これを使えば強正規性[8]を崩す例が作れる。

$$\mathsf{diverge}_T = \lambda \_ : \mathsf{Unit}.\ \mathsf{fix}_T\ (\lambda \mathsf{x} : \mathsf{T}.\ \mathsf{x})$$

この関数は Unit を受け取ると評価が止まらない[9]。

3.6 Untyped Lambda-Calculus, Redux

実は、型無しラムダ計算を再帰型を用いた型付きラムダ計算で表すことができる。初めに、対象とするラムダ計算は変数とラムダ抽象と関数適用しか無いと仮定する。この時に以下のような型 D と関数を定義する。

$$\begin{aligned} \mathsf{D} &= \mu \mathsf{X}.\mathsf{X} \to \mathsf{X} \\ \mathsf{lam} &= \lambda \mathsf{f} : \mathsf{D} \to \mathsf{D}.\ \mathsf{f}\ \mathsf{as}\ \mathsf{D} \\ \mathsf{ap} &= \lambda \mathsf{f} : \mathsf{D}.\ \lambda \mathsf{a} : \mathsf{D}.\mathsf{f}\ \mathsf{a} \end{aligned}$$

すると、以下のようにエンコーディングが可能になる。(左辺は型無し、右辺では型あり。このエンコーディングは実質的には何もしていない。)

[8] 正規性とは型付け出きる項は簡約が止まるという性質。

[9] $(\lambda \mathsf{x}\ \mathsf{x})(\lambda \mathsf{x}\ \mathsf{x})$ の評価のような感じ。

$$
\begin{aligned}
\mathsf{x}^* &= \mathsf{x} \\
(\lambda \mathsf{x}.\mathsf{M})^* &= \mathsf{lam}(\lambda \mathsf{x} : \mathsf{D}.\mathsf{M}^*) \\
(\mathsf{M}\ \mathsf{N})^* &= \mathsf{ap}\ \mathsf{M}^*\ \mathsf{N}^*
\end{aligned}
$$

また、これを拡張することもできる。ここでは、自然数を含むように拡張する例を示す。

$$
\begin{aligned}
\mathsf{D} &= \mu \mathsf{X}.\ \langle \mathsf{nat} : \mathsf{Nat}, \mathsf{fn} : \mathsf{X} \to \mathsf{X} \rangle \\
\mathsf{lam} &= \lambda \mathsf{f} : \mathsf{D} \to \mathsf{D}.\ \langle \mathsf{fn} = \mathsf{f} \langle\ \mathsf{as}\ \mathsf{D} \\
\mathsf{ap} &= \lambda \mathsf{f} : \mathsf{D}.\ \lambda \mathsf{a} : \mathsf{D}. \\
&\quad\ \mathsf{case}\ \mathsf{f}\ \mathsf{of} \\
&\quad\ \langle \mathsf{nat} = \mathsf{n} \rangle \Rightarrow \mathsf{diverge}_D\ \mathsf{unit} \\
&\quad\ \langle \mathsf{fn} = \mathsf{f} \rangle \Rightarrow \mathsf{f}\ \mathsf{a}
\end{aligned}
$$

関数適用において、与えられる f についての場合分けがされている[10]。このタグチェックは Scheme のような動的な型チェックをする言語の実装における実行時のタグチェックに非常に近いものがある。タグチェックが必要な関数例は D における「+1」関数が挙げられる。

$$
\begin{aligned}
\mathsf{suc} &= \lambda \mathsf{f} : \mathsf{D}.\ \mathsf{case}\ \mathsf{f}\ \mathsf{of} \\
&\quad\ \langle \mathsf{nat} = \mathsf{n} \rangle \Rightarrow (\langle \mathsf{nat} = \mathsf{succ}\ \mathsf{n}\ \rangle\ \mathsf{as}\ \mathsf{D}) \\
&\quad\ \langle \mathsf{fn} = \mathsf{f} \rangle \Rightarrow \mathsf{diverge}_D\ \mathsf{unit}
\end{aligned}
$$

ゼロの定義は明らか。

$$\mathsf{zro} = \langle \mathsf{nat} = 0 \rangle\ \mathsf{as}\ \mathsf{D}$$

4 Subtyping

サブタイプと再帰型を混ぜるとどうなるだろうか。例えば、Nat のサブタイプである Even を考えたとき、μX.Nat → (Even × X) と μX.Even → (Nat × X) の関係がどうなるかということである。これを簡単に考える方法は、再帰型を equi-recursive に扱うことである。上記の例では両方の型ともプロセス型と考えられる。前者の型に属するプロセスは任意の自然数を受け取って偶数を生み出す。後者の型に属するプロセスは任意の自然数を生み出すが偶数を与えられると期待されている。関数型のサブタイプを考えると直感的には前者は後者のサブタイプだと期待するがどうであろうか。正確な議論は 21 章で行う。

[10] この例では自然数の場合は評価が止まらないようになっているが、例外を出す実装もできるだろう。