Existential Types

水上 達夫

平成 13 年 10 月 1 日

前章では universal type ($\forall X.T$) を勉強したが、この章では existential type($\exists X.T$) について勉強する。どのように役に立つのかを先に言っておくと、これは Abstraction や Module System 等を形式的に議論するのに役立つ。

24.1 Motivation

Existential Type はプログラミング言語のモジュールや抽象データ型と密接に関係がある。この操作的な意味を強調するために、この本では型の Syntax を $\exists X.T$ から $\{\exists X, T\}$ に変更する。また、term $\{*S, t\}$ の $*$ は、tuple と混同しないように付加しているわけです。

Universal [**Type :** $\forall X.T$**, Term :** $\lambda X.t$]

論理　任意の型 S に対し、$[X \mapsto S]T$ の型を持つ値

直感　ある型 S が与えられると、$[X \mapsto S]T$ の型をもつ term を返す関数

Existential [**Type :** $\{\exists X.T\}$**, Term :** $\{*S, t\}$]

論理　ある型 S に対し $[X \mapsto S]T$ の型を持つ値

直感　型と term のペア (型 S を型 X で隠蔽したデータ t)

24.1.1 Packing

形式的な定義を見る前に、具体的に例を見てみることにしましょう。

```
p = {*Nat, {a=5, f=λ x:Nat. succ(x)}}
```

p の term 部分 (`{a=5, ...}`) は実際には

$$\{a : Nat, f : Nat \to Nat\} \tag{A}$$

の型を持ち、これの Nat 型を隠蔽すると

$$\{a : X, f : X \to X\} \tag{B}$$

になるので、p は $\{\exists X, \{a : X, f : X \to X\}\}$ を持つことがわかります。

今、(A) から (B) に型を変形しましたが、論理的に見ると、(B) の X という型に隠蔽されているデータ型 (Nat) を代入することで、実際にもっているデータ型 (A) に等しくなれば、p は (B) の型をもつことが許されるわけです。

一方、p は $\{\exists X, \{a : X, f : X \to \mathrm{Nat}\}\}$ の型も持ってもいいかと思います。これでは p に一意な型を付けることができないので、昔やった Ascription という注釈をつけます。

```
  p = {*Nat, {a=5, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→X}}
> p : {∃X, {a:X, f:X→X}}
  p2 = {*Nat, 0} as {∃X, X}
> p2 : {∃X, X}
  p3 = {*Bool, true} as {∃X, X}
> p3 : {∃X, X}
```

これらの形式的な定義は次のようになります。

$$\frac{\Gamma \;\vdash\; t_2 : [\mathrm{X} \mapsto \mathrm{U}]T_2}{\Gamma \;\vdash\; \{*\mathrm{U}, t_2\} \text{ as } \{\exists X, T_2\} : \{\exists X, T_2\}} \qquad \text{(T-PACK)}$$

では、もっと有益な例 (データ抽象したオブジェクト) を作成してみることにします。

```
  p4 = {*Nat, {a=0, f=λx:Nat. succ(x)}} as {∃X, {a:X, f:X→Nat}}
  p5 = {*Bool, {a=true, f=λx:Bool. 0}} as {∃X, {a:X, f:X→Nat}}
```

p4，p5 が同じ型`{∃X, {a:X, f:X→Nat}}` を持ち、それぞれの実装データなんて気にせずに λ 適用で f a : Nat なんてことができるわけです。また、気をつけなくてはならないのは、最初の例 p において、

```
  p6 = ... as {∃X, {a:X, f:X→X}}
  p7 = ... as {∃X, {a:X, f:Nat→X}}
  p8 = ... as {∃X, {a:Nat, f:Nat→Nat}}
```

と型付けを行っても、抽象データ型として役に立たないということです。p6 はどのような操作を行っても、型 X 以外の要素を得ることができず、その要素を他に使いようがありません。p7 に至っては、f a と適用すらできません。また、p8 は抽象化する意味がありません。

評価規則　Packing の評価規則は term 部分が value になるまで評価するために、次のようになります。

$$\frac{t_{12} \to t'_{12}}{\{*T_{11}, t_{12}\} \text{ as } T_1 \to \{*T_{11}, t'_{12}\} \text{ as } T_1} \qquad \text{(E-PACK)}$$

24.1.2 Unpacking

上の節では f a と適用すると適当に言いましたが、例えばオブジェクト (モジュール)p4 の f と a を使用するには、上の方法ではもちろん駄目で、形式的に定義にきちんと定義してやる必要があります。

$$\frac{\Gamma \ \vdash \ t_1 : \{\exists \mathrm{X}, T_{12}\} \qquad \Gamma, \mathrm{X}, x : T_{12} \ \vdash \ t_2 : T_2}{\Gamma \ \vdash \ \mathrm{let}\ \{\mathrm{X}, x\} = t_1 \ \mathrm{in}\ t_2 : T_2} \qquad \text{(T-UNPACK)}$$

オブジェクト t_1 の X で抽象化されたインターフェース (型) を x に取り出し (バインドし)、その環境下で t_2 を型付けしています。例えば、

`let {X, x} = p4 in (x.f x.a)`

は、p4 は`{∃X, {a:X, f:X→Nat}}` に型付けされているので、x は`{a:X, f:X→Nat}}` に型付けされ、`let ... in (x.f x.a) : Nat` となります。

間違い Packing と同様に間違いの例を見てみることにします。

- (抽象性) `let {X, x} = p4 in succ(x.a)`
  たとえ、x.a の実装が Nat 型であっても、x.a は X で抽象化されているために、これは型エラーとなります
- (スコープ) `let {X, x} = p4 in x.a`
  x.a は型 X であり、型 X は let スコープから出てはいけません。よって、
  `let {X, y} = p4' in y.f (let {X, x} = p4 in x.a)`
  も駄目です。この制限がないと、x, y の見かけ上の型は同じであるが、y.f (let 文) を評価する際に、型 X が抽象している実際の型が場所によって異なることが生じてしまいます。

評価規則 最後に評価規則ですが、パッケージが作成するための規則

$$\frac{t_1 \to t_1'}{\mathrm{let}\ \{X, x\} = t_1 \ \mathrm{in}\ t_2 \to \mathrm{let}\ \{X, x\} = t_1' \ \mathrm{in}\ t_2} \qquad \text{(E-UNPACK)}$$

と、パッケージを Unpack して、body を評価する規則

$$\mathrm{let}\ \{X, x\} = (\{*T_{11}, v_{12}\} \ \mathrm{as}\ T_1) \ \mathrm{in}\ t_2 \to [X \mapsto T_{11}][x \mapsto v_{12}]t_2 \qquad \text{(E-UNPACKPACK)}$$

が存在します。

24.1 Data Abstraction with Existentials

抽象化すると何がいいことがあるのかは米研メンバーなら自明でしょう。よって、ここでは抽象化するための方法として、ADT, Existential Object, Hybrid Object を述べ、その説明と特色のみ述べることにします。

24.1.1 ADT

Abstract data type (ADT) は型変数 type A, 実際の型 representation T, それに対する operations で構成され、ADT の内部からは T が見えるが、ADT の外部からは T の代わりに A が見える。具体的な定義方法は下記のようになる。

```
ADT counter =
  type Counter
  representation Nat
  signature
    new : Counter,
    get : Counter -> Nat,
    inc : Counter -> Counter;
  operations
    new = 1,
    get = λi:Nat. i,
    inc = λi:Nat. succ(i);
```

ADT と先ほどのパッケージとの違いは、パッケージは実際のデータを格納するフィールドと関数テーブルを持ち合わせていたが、ADT においては実際のデータしか持ち合わせていない。

また、ADT は Existential Type にエンコードすることができる。例えば、上の ADT をエンコードすると次のようになる。

```
# counterADT =
    {*Nat,
      {new = 1,
       get = λi:Nat. i,
       inc = λi:Nat. succ(i)}}
    as {∃Counter,
      {new: Counter,
       get: Counter -> Nat,
       set: Counter -> Counter}};
> counterADT : {∃Counter,
    {new:Counter, get:Counter -> Nat, inc:Counter -> Counter}}
# let {Counter, counter} = counterADT in
    counter.get (counter.inc counter.new);
> 2 : Nat;
```

また、以前に定義した ADT で別の ADT を定義するには、次のように使用する。

```
  let {Counter, counter} = counterADT in
```

```
let {FlipFlop, flipflop} =
  {*Counter,
    {new   = counter.new,
     read  = λc:Counter. iseven (counter.get c),
    ....
```

24.1.2 Existential Object

この章の最初に説明したパッケージとほぼ同じである。関数テーブルの意味を強調するために、次のように関数テーブル用のフィールドを別に用意している。

```
  c = {*Nat,
        {state = 5,
         methods = {get = λx:Nat. x,
                    inc = λx:Nat. succ(x)}}}
      as Counter;
```

ただし、 `Counter = {λX, {state:X, methods: {get:X -> Nat, inc:X -> X}}};`
purely functional program のために、カウンタの値を 1 増やすためには、次のように新たにオブジェクトを作成しなくてはならない。

```
  sendinc = λc:Counter.
     let {X, body} = c in
       {*X,
         {state = body.methods.inc(body.state),
          methods = body.methods}}
       as Counter
```

この抽象化の方法は、オブジェクト指向言語の利点を多大に受けることができ、サブタイプや継承の導入もスムーズにできる。

しかし、オブジェクトに対するすべての操作を抽象化されたインターフェース (型) をもとにしなくてはならないために、現実のプログラミングには制限がきつい。例えば、Java, C++ライクの次のようなことができない。

```
class C {
    private int a;
    public int sum(C obj) {
          return a + obj.a;
    }
    public int getValue() {
        return a;
    }
}
```

なぜなら、Existential Object においては、同じ抽象型でも異なる実装を持ちうることがあるからである。(obj.a は int ではなく、Integer Object かもしれない)
一方、ADT においては上のことが可能である。

24.1.3 Hybrid Object

上記の問題を解決するために、現実主流のプログラミング言語においては Object と ADT の妥協点を採用している。これらの言語では、オブジェクトの型はクラス名であり、たとえ同じインターフェースのクラスがあっても、先ほどとは異なる型としている。サブタイプが加わると、ある型 A のサブタイプ B があると、クラス B は必ずクラス A から継承を行う必要があり、その際にフィールドをすべて子クラス B にも引き継ぐように定義している。
こうすることで、クラス C の型と定義されたオブジェクトは必ず int 型のフィールド a を持つことが保証されるわけである。

24.1 Encoding Existentials

この節では Existential Type が前章で説明した Universal Type でエンコードできることを説明する。

- packing
  Existential Type の型 $\{\exists X, T\}$ をよく見てもらうと、これは Pair Type であることがわかる。では、Pair Type が Unisersal Type ではどのように型付けされたのかを見るために、まず整数がどのように型付けされるのかを見てみることにする。$\lambda_{\to}$ の Church Encoding では

  $$C_n = \lambda s.\lambda z.\, \underbrace{s(\cdots(s(s}_{n}\, z))\cdots)$$

  整数 n は、z の型を受け取ると、その型の値を返す関数 s と初期値 z を受け取ると、s を n 回適用する関数と定義したわけである。すなわち、整数 n の型は

  $$\text{CNat} = \forall X.(X \to X) \to X \to X$$

  と System F では型付けできたわけである。一つ注意すべきことは、上の Church Encoding が整数 n を表す唯一の λ 式ではないということである。このことは、以降の Pair, Packing についてもいえることである。では、Pair Type はどのように型付けできたのか？もっと、問題を簡単にするために Nat の値しかもち得ない Pair Type (PairNat) について考える。PairNat の値 (a, b) があると、関数 $f(a, b)$ のようにエンコー

ディングできないかと考えると、実際それができて前章では次のように型付けできたのである。

$$\mathrm{PairNat} = \forall X.(\mathrm{CNat} \to \mathrm{CNat} \to X) \to X$$

では、Existential Type $\{\exists X, T\}$ を Pair と考えると

$$\{\exists X, T\} = \forall Y.(\forall X.T \to Y) \to X$$

はならないだろうかと考えれるわけである。実際、この型の値 $\{*S, t\}$ as $\{\exists X, T\}$ は Universal Type では次のようにエンコーディングできる。

$$\{*\mathrm{S}, \mathrm{t}\} \text{ as } \{\exists X, T\} \equiv \forall Y.\lambda \mathrm{f} : (\forall X.T \to Y).\ \mathrm{f}\ [\mathrm{S}]\ \mathrm{t}$$

まさに、f [S] t の部分に実際の型とデータが隠蔽されているのである。言葉に言い直すと、∀ Y で、オブジェクトに対する操作の結果の型、λf で、抽象した型 X とオブジェクトの型 T に関する関数を Unpack するときに使用すると、それらを具体化して返すことを行う。そして、この Packing のエンコードに対する、Unpacking のエンコードをきちんと定義できれば、Existential Type の Universal Type によるエンコーディングが完成する。

- unpacking
  let $\{X, x\} = t_1$ in $t_2 : \{\exists X.T_{11}\}$ をエンコードすることを考える。
  $t_1 = \{*T_{11}, v_{12}\}$ as $\{\exists X, T\}$ とおくと、t_1 は次のようにエンコードできた。

  $t_1 = \forall \mathrm{Y}.\lambda \mathrm{f} : (\forall \mathrm{X}.T \to Y).\mathrm{f}\ [T_{11}]\ v_{12}$

  t_1 に let 文の返す型 T_2、let 文の返す値を t_1 の引数の型と同じになるようにうまく λ 抽象化したものをすなおに t_1 に適用したものを U とすると、

$$\begin{aligned} U \equiv\ & t_1[T_2](\lambda X.\lambda x : T_{12}.t_2) \\ =\ & \lambda f : (\forall X.T \to Y).(f[T_{11}]v_{12})(\lambda X.\lambda x : T_{11}.t_2) \\ =\ & (\lambda X.\lambda x : T_{12}.t_2)[T_{11}]v_{12} \\ =\ & (\lambda X.\lambda x : T_{12}.t_2)[T_{11}]v_{12} \\ =\ & ([X \mapsto T_{11}](\lambda x : T_{12}.t_2))v_{12} \\ =\ & ([X \mapsto T_{11}]((\lambda x : T_{12}.t_2)v_{12}) \\ =\ & [X \mapsto T_{11}][x \mapsto v_{12}]t_2 \end{aligned}$$

  となり、この結果は (E-UNPACKPACK) の結果と一致する。というわけで、U すなわち、let $\{X, x\} = t_1$ in t_2 は Universal Type を使用して次のようにエンコーディングできる。

$$\text{let } \{X, x\} = t_1 \text{ in } t_2 \equiv t_1\ [T_2]\ (\lambda X.\lambda x : T_{11}.t_2)$$