# Abstract Types have Open Existential Types

El Pin Al

型 Advent Calendar 2019 1 日目の記事です。

存在量化は項の型を、その型の構造の一部を隠すことによって抽象化します。System F での存在型の導入、除去の型付け規則は次の通りです。

$$\frac{\Gamma \vdash M : \tau'[\alpha \leftarrow \tau]}{\Gamma \vdash \texttt{pack}\ \langle \tau, M \rangle\ \texttt{as}\ \exists\alpha.\tau' : \exists\alpha.\tau'}\ \text{PACK}$$

$$\frac{\Gamma \vdash M : \exists\alpha.\tau \qquad \Gamma, \alpha, x : \tau \vdash M' : \tau' \qquad \alpha \notin \text{ftv}(\tau')}{\Gamma \vdash \texttt{unpack}\ M\ \texttt{as}\ \alpha, x\ \texttt{in}\ M' : \tau'}\ \text{UNPACK}$$

pack は項 $M$ の型内部での型 $\tau$ の出現を $\exists\alpha.\tau'$ に沿って隠すことで存在型を導入します。unpack は、項 $M$ が存在型 $\exists\alpha.\tau$ を持つとき、その存在量化子を除去します。項 $M'$ は $M$ の (hypothetical な) 証人型 $\alpha$ と値変数 $x$ にアクセスできます。また、$\alpha$ がそのスコープから脱出しないように、$\alpha \notin \text{ftv}(\tau')$ という条件が付いています。

Mitchell と Plotkin [1] は抽象型は存在型として理解できることを示しましたが、同時に、存在型はモジュールにおける型抽象 (特に型生成性の概念) を正確にモデル化しないこと、それとモジュラリティの欠如にも気付きました。

System F の pack, unpack にはモジュラリティ上の問題があります。

pack の問題は冗長性です。pack では、結果の型を完全に指定しないといけないので、抽象化が必要ない部分の型情報を重複させてしまいます。これは「存在量化の導入」と「型のどの部分を隠すかの指定」が分離されていないことに起因します。

unpack の問題は非局所性です。unpack $M$ as $\alpha, x$ in $M'$ は型変数 $\alpha$ と値変数 $x$ が同じスコープにあることを強制します。これにより unpack された項の利用は全て予期されていなければなりません。$\alpha$ をプログラム全体で使う唯一の方法は unpack をプログラムの十分早い位置に置くことです。これは非局所的な (つまりモジュラーではない) プログラム変換です。この問題の原因は unpack が一度に多くのことをやりすぎているからです。

どちらの場合も、モジュラリティの欠如は pack, unpack の原子性の欠如に由来します。存在型の表現力を保ちつつ、モジュラリティを向上させるために、Montagu と Rémy [2] は存在型の導入、除去をより細かい構成子に分解した open existential type を提案しました。

Montagu と Rémy は System F に open existential type を追加した$F^{\curlyvee}$ (F-zip と読む) という言語を作りました。

Open existential type の論文は 4 本 [2][3][4][5] ありますが、この文書ではその中でも最初に出た [2] の紹介をします。

# 1. Open existential types

## 1.1. 型付け環境

存在型の分解をするために、まず型付け判断の環境を拡張します。

System F においては、型付け環境は $x : \tau$ と $\alpha$ の列です。

$F^{\curlyvee}$では型付け環境を 2 つの新しい要素で拡張します。1 つ目は**存在型変数** $\exists\alpha$ といい、抽象型のスコープを追跡するためにあります。この存在型変数と区別するために、System F での $\alpha$ は $\forall\alpha$ と書き、**全称型変数**と呼びます。もう 1 つは**型定義** $\alpha = \tau$です。型定義は型の抽象的な見方 $\alpha$ と具象的な見方 $\tau$の間を簡潔に仲介するためにあります。型付け環境は結果的に次のようになります。

$$
\begin{array}{ll}
\Gamma ::= \varepsilon \mid \Gamma, b & \text{(環境)} \\
\theta ::= \alpha & \\
b ::= x : \tau \mid \forall\theta \mid \theta = \tau \mid \exists\theta & \text{(Bindings)}
\end{array}
$$

ここで出てくる $\theta$ は、今は単なる型変数ですが、後の拡張のために用意しています。環境の well-formedness (元論文参照) は同じ変数を 2 度束縛しないことを保証します。

## 1.2. pack の分解

次は pack を 2 つの構成子に分解します。

1 つ目は**存在導入** (existential introduciton) です。存在導入 $\exists(\alpha = \tau)M$ は、$M$ を型付けるときに存在型変数 $\alpha$ を証人型 $\tau$ と共に環境に導入し、最終的に $\alpha$ を存在量化します。

$$
\frac{\Gamma, \alpha = \tau \vdash M : \tau'}{\Gamma \vdash \exists(\alpha = \tau)M : \exists\alpha.\tau'}\ \text{EXISTS}
$$

もう 1 つは**強制** (coercion) です。強制 $(M : \tau)$ は $M$ の型を compatible な型で置き換えます。Compatibility 関係 $\approx$ は環境に現れる全ての型定義を含んだ最小の合同関係です。

$$\frac{\Gamma \vdash M : \tau' \qquad \Gamma \vdash \tau' \approx \tau}{\Gamma \vdash \big(M : \tau\big) : \tau}\ \text{COERCE}$$

存在導入と強制を用いて、pack を定義することができます。

$$\texttt{pack}\ \langle \tau, M \rangle\ \texttt{as}\ \exists\alpha.\tau' \triangleq \exists(\alpha = \tau)\big(M : \tau'\big)$$

型の「どの部分」が隠されるかの指定 (強制によって行われる) が、実際の隠蔽操作 (存在導入によって行われる) とは分離されることによって、型情報の重複を防ぐことができます。pack の分解によって、存在型の値の作成がより簡潔になり、管理しやくなります。

## 1.3. unpack の分解

pack の次は unpack を 2 つの構成子に分解します。

1 つ目は opening です。Opening $\texttt{open}^{\alpha}\ M$ は存在型 $\exists\alpha.\tau$ を持つ項 $M$ を $\alpha$ を用いて開きます。$\alpha$ は環境で $\exists\alpha$ として追跡されます。Opening は存在型を持つ項のスコープのない unpack を実現します。

$$\frac{\Gamma \vdash M : \exists\alpha.\tau \qquad \alpha \notin \operatorname{dom}\Gamma}{\Gamma, \exists\alpha \vdash \texttt{open}^{\alpha}\ M : \tau}\ \text{OPEN}$$

この型付け規則を下から上に読むと、存在型変数 $\exists\alpha$ は opening によって消費される線形な資源として見ることができます。

もう 1 つは restriction です。Restriction $\nu\alpha.M$ は、unpack の「型変数は脱出してはならない」という条件に対応します。Restriction は $M$ の型内部に $\alpha$ の自由な出現がないことを要請することによって、抽象型 $\alpha$ のスコープを制限します。そして「$M$ の部分項として現れる $\texttt{open}^{\alpha}\ M'$」によって消費されなければならない存在型変数 $\exists\alpha$ を環境に入れます。

$$\frac{\Gamma, \exists\alpha \vdash M : \tau \qquad \alpha \notin \operatorname{ftv}(\tau)}{\Gamma \vdash \nu\alpha.M : \tau}\ \text{NU}$$

pack と同じように、unpack も糖衣構文として定義できます。

$$\texttt{unpack}\ M\ \texttt{as}\ \alpha, x\ \texttt{in}\ M' \triangleq \nu\alpha.\big(\texttt{let}\ x\ \texttt{=}\ \texttt{open}^{\alpha}\ M\ \texttt{in}\ M'\big)$$

unpack が $\nu$, let, open の組み合わせから成るということは、unpack が 3 つの操作を同時に行なっていたことを示唆します。

unpack の分解によって、抽象型 $\alpha$ はプログラムの最も外側、または型付け文脈で導入する

ことができます。これはモジュールにおける型抽象を綿密にモデル化します。

## 1.4. 線形性

同じ存在型変数を用いて異なる存在型の値を open することを許すと、型システムが unsound になります。

$$\begin{aligned}
&\texttt{let } f \texttt{ = open}^{\alpha}\ \exists\big(\beta = \texttt{int}\big)\big(\lambda z : \texttt{int}.z + 1\big) : \beta \to \beta \texttt{ in}\\
&\texttt{let } x \texttt{ = open}^{\alpha}\ \exists\big(\beta = \texttt{bool}\big)\ \texttt{true} : \beta \texttt{ in}\\
&f\ x
\end{aligned}$$

ここでは、存在型変数 $\alpha$ が 2 回の opening で利用されています。この式は最終的に $\texttt{true} + 1$ に評価されてしまいます。この問題を防ぐには、各存在型変数 $\alpha$ が丁度 1 回だけ opening によって消費されるようにすればよいです。型付け環境の存在型変数を線形に扱うために、環境の zipping という操作を考えます。Zipping は対称的なバージョン $(\cdot \curlyvee \cdot)$ と非対称的なバージョン $(\cdot \curlywedge \cdot)$ があり、次のように定義されます。

$$\begin{aligned}
\varepsilon \curlyvee \varepsilon &= \varepsilon\\
\big(\Gamma_1, b\big) \curlyvee \big(\Gamma_2, b\big) &= \big(\Gamma_1 \curlyvee \Gamma_2\big), b && \text{if } b \neq \exists\alpha\\
\big(\Gamma_1, \exists\alpha\big) \curlyvee \Gamma_2 &= \big(\Gamma_1 \curlyvee \Gamma_2\big), \exists\alpha && \text{if } \alpha \notin \operatorname{dom}\Gamma_2\\
\Gamma_1 \curlyvee \big(\Gamma_2, \exists\alpha\big) &= \big(\Gamma_1 \curlyvee \Gamma_2\big), \exists\alpha && \text{if } \alpha \notin \operatorname{dom}\Gamma_1
\end{aligned}$$

$$\begin{aligned}
\varepsilon \curlywedge \varepsilon &= \varepsilon\\
\big(\Gamma_1, b\big) \curlywedge \big(\Gamma_2, b\big) &= \big(\Gamma_1 \curlywedge \Gamma_2\big), b && \text{if } b \neq \exists\alpha\\
\big(\Gamma_1, \exists\alpha\big) \curlywedge \Gamma_2 &= \big(\Gamma_1 \curlywedge \Gamma_2\big), \exists\alpha && \text{if } \alpha \notin \operatorname{dom}\Gamma_2\\
\Gamma_1 \curlywedge \big(\Gamma_2, \exists\alpha\big) &= \big(\Gamma_1 \curlywedge \Gamma_2\big), \exists\alpha && \text{if } \alpha \notin \operatorname{dom}\Gamma_1\\
\big(\Gamma_1, \exists\alpha\big) \curlywedge \big(\Gamma_2, \forall\alpha\big) &= \big(\Gamma_1 \curlywedge \Gamma_2\big), \exists\alpha
\end{aligned}$$

## 1.5. 再帰型の出現

unpack の分解は再帰型への道を切り開いてしまいます。なぜなら、抽象型変数の証人型が定義される前に、その変数を利用できるからです。再帰型は opening を通して、2 種類の方法で現れることができます。

1 つ目は内部再帰性 (internal recursivity) と呼ばれます。

$$\texttt{let}\ x = \exists(\alpha = \beta \to \beta) M\ \texttt{in}\ \texttt{open}^{\beta}\ x$$

抽象型変数 $\beta$ は、$x$ を定義する際に証人型として使われていますが、$x$ は $\beta$ の名前を利用して開かれています。これを簡約することで、次の項が得られます。

$$\texttt{open}^{\beta}\ \exists(\alpha = \beta \to \beta) M$$

これは $\beta = \beta \to \beta$ という再帰的な等式に至ります。このような opening の利用を避けるために、let に対しては非対称的な zipping $(\cdot\curlyvee\cdot)$ を使います。$\texttt{let}\ x = M_1\ \texttt{in}\ M_2$ において、$M_2$ で存在型変数が利用される場合、 対応する型変数は $M_2$ で利用することはできません。これにより let 式における抽象型変数の早すぎる利用を回避します。

もう 1 つは**外部再帰性** (external recursivity) です。

$$\left\{l_1 = \texttt{open}^{\beta_1}\ \exists\left(\alpha_1 = \beta_2 \to \beta_2\right) M_1;\ l_2 = \texttt{open}^{\beta_2}\ \exists\left(\alpha_2 = \beta_1 \to \beta_1\right) M_2\right\}$$

このコードでは証人型が相互に定義されています。もし型抽象を取り除いたら、再帰的な等式系 $\beta_1 = \beta_2 \to \beta_2$ と $\beta_2 = \beta_1 \to \beta_1$ が手に入ります。内部再帰性のときと同じように、非対称な zipping を使う手もありますが、そのようにすると、非対称な簡約戦略を強制してしまいます。代わりに対称的な zipping を、 レコードや関数適用などには使います。もし、 存在型変数を 1 つの部分項で使う場合、 別の部分項ではその変数を一切使うことができないようになります。この制約は割と厳しくて、(最低でも System F 程度の表現力はありますが、) [3] などでより制限の緩い型システムが提案されています。

## 2. F$^{\curlyvee}$の構文

F$^{\curlyvee}$は System F にレコードと open existential type を追加した言語です。Open existential type は新しい形式の型を一切導入しないので、型の文法は System F と同じになります。

$$\begin{aligned}
\tau &::= \alpha \mid \tau \to \tau \mid \{l_1 : \tau_1,\ ...,\ l_n : \tau_n\} \mid \forall\theta.\tau \mid \exists\theta.\tau \\
M &::= x \mid \lambda x : \tau.M \mid M\ M \mid \texttt{let}\ x = M\ \texttt{in}\ M \mid \Lambda\theta.M \mid M[\tau] \\
&\quad \mid \exists(\theta = \tau) M \mid (M : \tau) \mid \nu\theta.M \mid \texttt{open}^{\alpha}\ M \mid \{r\} \mid M.l \\
r &::= \varepsilon \mid l = M\ ;\ r \mid l : \tau = M\ ;\ r
\end{aligned}$$

さらに、以下の糖衣構文を導入します。

$$\Sigma^{\beta}(\alpha = \tau) M \triangleq \texttt{open}^{\beta}\ \exists(\alpha = \tau) M$$

## 3. 型付け規則

Open existential type のための型付け規則はもう既に述べたので、それ以外の規則の一部を載せます。($\Gamma \vdash$ wf は環境の well-formedness を検査します。詳しい定義は元論文を参照のこと。)

$$\frac{\Gamma \vdash \text{wf} \qquad \Gamma\ pure}{\Gamma \vdash x : \Gamma(x)}\ \text{VAR} \qquad \frac{\Gamma_1 \vdash M_1 : \tau' \to \tau \qquad \Gamma_2 \vdash M_2 : \tau'}{\Gamma_1 \curlyvee \Gamma_2 \vdash M_1\ M_2 : \tau}\ \text{APP}$$

$$\frac{\Gamma_1 \vdash M_1 : \tau_1 \qquad \Gamma_2, x : \tau_1 \vdash M_2 : \tau_2}{\Gamma_1 \curlyvee \Gamma_2 \vdash \texttt{let}\ x\ \texttt{=}\ M_1\ \texttt{in}\ M_2 : \tau_2}\ \text{LET}$$

$$\frac{\Gamma' \vdash M : \tau \qquad \Gamma \Vdash \Gamma'}{\Gamma \vdash M : \tau}\ \text{SHIFT} \qquad \frac{\Gamma, \forall\alpha \vdash M : \tau \qquad \Gamma\ pure}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\tau}\ \text{GEN}$$

Environment entailment $\Gamma \Vdash \Gamma'$ は、SHIFT 規則で存在型変数を環境の右側に写すために用いられます:

$$\frac{}{\Gamma \Vdash \Gamma} \qquad \frac{\Gamma_1 \Vdash \Gamma_2 \qquad \Gamma_2 \Vdash \Gamma_3}{\Gamma_1 \Vdash \Gamma_3} \qquad \frac{\alpha \notin \text{ftv}(b)}{\Gamma_1, \exists\alpha, b, \Gamma_2 \Vdash \Gamma_1, b, \exists\alpha, \Gamma_2}$$

SHIFT 規則の必要性は、OPEN 規則が環境の右端に存在型変数を置くことを要求していることに起因します。

APP 規則は対称的な zipping を、LET 規則は非対称的な zipping を使っています。

型付け規則は、 線形性を破壊することなく値を代入できることを保証しなければなりません。 環境が存在型変数を含んでいないとき、 線形性を破壊せずに済みます。$\Gamma$ がそのような環境であるとき、$\Gamma$ は純粋であるといい、$\Gamma\ pure$ と書きます。VAR 規則や GEN 規則で $\Gamma\ pure$ という条件があるのは、それらが値であるからです。

## 4. 意味論

$\text{F}^{\curlyvee}$ は小ステップの値呼び簡約意味論を採用しています。

$\text{F}^{\curlyvee}$ において、 任意の項を安全に代入することはできません。 代入は opening の線形性を侵害し得るからです。純粋な環境で型付け可能な項を、純粋な項と呼ぶと、そのような項は安全に代入できます。 したがって、 それ以上簡約できない項で、 行き詰まり状態でないものの内、 純粋であるものを値と呼ぶことにします。$\Sigma^{\beta}(\alpha = \tau)\lambda x : \alpha.x$ のような項は、 環境に存在型変数 $\exists\beta$ がないといけないので、 純粋ではありませんが、 これ以上簡約できず、

しかも行き詰まり状態ではありません。このような項は代入することができません。この問題は $\Sigma$ を十分外側に押し出して、簡約を進められるようにすることによって解決します。

$$
\begin{aligned}
&\texttt{let } x = \Sigma^{\beta}(\alpha = \texttt{int})(1 : \alpha) \texttt{ in } \left(\lambda y : \beta.y\right) x \\
&\rightsquigarrow \Sigma^{\beta}(\alpha = \texttt{int}) \texttt{ let } x = (1 : \alpha) \texttt{ in } \left(\lambda y : \alpha.y\right) x \\
&\rightsquigarrow \Sigma^{\beta}(\alpha = \texttt{int}) \left(\lambda y : \alpha.y\right) (1 : \alpha) \\
&\rightsquigarrow \Sigma^{\beta}(\alpha = \texttt{int}) (1 : \alpha)
\end{aligned}
$$

この例の最初の簡約時に、$\Sigma$ は `let` の外側に押し出されています。$\Sigma$ がより広いスコープを持つようになったので、$\lambda$ 抽象における $\beta$ が $\alpha$ で置き換えられています。この置換がないと subject reduction が成り立たなくなってしまいます。

## 5. 健全性

健全性は subject reduction と進行によって証明されます。Subject reduction の証明はまったくもって標準的であり、ほとんど単純である、と Montagu と Rémy は主張しています。これは他のモジュールシステムに対する優位点です。進行も型付け導出の帰納法で証明できます。

## 6. 型レベル Path システム

$F^{\curlyvee}$ に、open existential type とは直行する拡張として、型レベルの path システムを導入します。型レベルの path システムは、type shape と type projection から成ります。

$$
\begin{aligned}
&\kappa ::= \star \mid \kappa \to \kappa \mid \left\{l_1 : \kappa_1, \ ..., \ l_n : \kappa_n\right\} && \text{(カインド)} \\
&\theta ::= \alpha \in \sigma \\
&\sigma ::= \top \mid \tau \mid [\theta]\sigma && \text{(Shapes)} \\
&\tau ::= ... \mid \tau.l \mid \tau.1 \mid \tau.2 && \text{(Projections)}
\end{aligned}
$$

$\star$ は基本となるカインド、$\kappa \to \kappa$ は関数型のカインド、$\left\{l_1 : \kappa_1, \ ..., \ l_n : \kappa_n\right\}$ はレコード型のカインドとなります。この path システムにおける $\star \to \star$ は $F_{\omega}$ などにおけるそれとは意味が違うので注意してください。

Shape は型の集合を表します。束縛型変数の範囲を制限するために使われます。Shape の解釈は以下の通りです。($\Gamma \vdash \tau :: \kappa$ は型 $\tau$ が環境 $\Gamma$ でカインド $\kappa$ を持つ、という意味で

す。詳細は元論文を参照のこと。)

$$\frac{\Gamma \vdash \tau :: \kappa}{\Gamma \vdash \tau \in \top} \qquad \frac{\Gamma \vdash \tau :: \kappa}{\Gamma \vdash \tau \in \tau} \qquad \frac{\Gamma \vdash \tau \in \sigma \qquad \Gamma, \forall(\alpha \in \sigma) \vdash \tau' \in \sigma'}{\Gamma \vdash \tau'[\alpha \leftarrow \tau] \in [\alpha \in \sigma]\sigma'}$$

Top shape $\top$ は全ての (well-formed な) 型の集合です。Singleton shape $\tau$ は型 $\tau$ のみを含む集合です。Compound shape $[\alpha \in \sigma]\sigma'$ は $\tau'[\alpha \leftarrow \tau]$ の集合です (ただし $\tau' \in \sigma'$ かつ $\tau \in \sigma$ とします)。

Singleton shape と type projection の存在は型同値性の概念を示唆します。

$$\frac{(\alpha \in \tau) \in \Gamma}{\Gamma \vdash \alpha \equiv \tau}\ \text{EQUI-VAR} \qquad \frac{\Gamma \vdash \tau \equiv \{..., l_k : \tau_k, \ ...\}}{\Gamma \vdash \tau.l_k \equiv \tau_k}\ \text{EQUI-PROJ-LABEL}$$

$$\frac{\Gamma \vdash \tau \equiv \tau_1 \to \tau_2}{\Gamma \vdash \tau.i \equiv \tau_i}\ \text{EQUI-PROJ-ARROW}$$

$$\frac{\alpha \notin \text{ftv}\big(\Gamma', \tau, \tau'\big) \qquad \Gamma, \forall(\alpha \in \sigma), \forall\big(\alpha' \in \sigma'\big), \Gamma' \vdash \tau \equiv \tau'}{\Gamma, \forall\big(\alpha' \in [\alpha \in \sigma]\sigma'\big), \Gamma' \vdash \tau \equiv \tau'}\ \text{EQUI-FOLD}$$

Shape と型同値性の追加に対応して、型付け規則にも変更を加えます。

$$\frac{\Gamma \vdash M : \forall(\alpha \in \sigma).\tau' \qquad \Gamma \vdash \tau :: \kappa \qquad \Gamma \vdash \tau \in \sigma}{\Gamma \vdash M[\tau] : \tau'[\alpha \leftarrow \tau]}\ \text{INST}$$

$$\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash \tau \equiv \tau'}{\Gamma \vdash M : \tau'}\ \text{EQUIV}$$

INST 規則では、$\tau$ として利用できるのは、$\sigma$ の要素である型だけになっています。EQUIV 規則は、型同値性を型付け規則に反映したものです。

型レベルの path システムは、 型をより簡潔かつモジュラーに書くことを可能にします。Shape と projection の例を 1 つ挙げておきます。

$$\begin{aligned}
&\forall\Big(\alpha_1 \in \Big[\beta_1 \in \top\Big]\Big[\beta_1' \in \top\Big]\Big\{l : \beta_1; \, l' : \beta_1'\Big\}\Big).\\
&\forall\Big(\alpha_2 \in \Big[\beta_2' \in \top\Big]\Big\{l : \alpha_1.l; \, l' : \beta_2'\Big\}\Big).\\
&\alpha_1 \to \alpha_2 \to \Big\{l : \alpha_1.l; \, l' : \alpha_1.l\Big\}
\end{aligned}$$

これは OCaml での次のようなファンクタシグネチャとおおよそ対応します。

```
module type S = functor (X : sig
  type a
  type b
  val l : a
  val l' : b
end) -> functor (Y : sig
  type b
  val l : X.a
  val l' : b
end) -> sig
  val l : X.a
  val l' : X.a
end
```

Shape は ML モジュールのシグネチャのような役割を果たし、projection は、引数間の型の sharing を表現します。

## 7. 参考文献


[1] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. 1988.

[2] Benoît Montagu and Didier Rémy. Towards a simpler account of modules and generativity: Abstract types have open existential types. 2008.

[3] Benoît Montagu and Didier Rémy. A logical account of type generativity: Abstract types have open existential types. 2008.

[4] Benoît Montagu and Didier Rémy. Modeling abstract types in modules with open existential types. 2009.

[5] Benoît Montagu. Programming with first-class modules in a core language with subtyping, singleton kinds and open existential types. 2010.