ソフトウェア基礎論配布資料 (6)
パラメタ多相と System F

五十嵐 淳
京都大学 大学院情報学研究科知能情報学専攻
e-mail: igarashi@kuis.kyoto-u.ac.jp

平成 19 年 12 月 11 日

1 多相性，パラメタ多相と System F

- 計算機科学一般における**多相性**(*polymorphism*)…ひとつの定義・操作・記号を，複数の異なる「種類」(「型」) のものとして使うことを許している，という言語 (多くの場合，プログラミング言語) に備わった性質.
- **多相的型システム**(*polymorphic type system*): 多相性を考慮にいれた型システム
- Strachey による polymorphism の 3 タイプの分類:
  - **ad-hoc polymorphism**(**アドホック多相**): 対象の適用範囲は複数の「型」に及ぶが，型の種類によって行なわれる実際の操作が異なる．通常，その適用範囲に必然性があまりない (ad-hoc). (例) C 言語における + は `int` の加算演算子であり，`float` の加算演算子でもあるが，実際の加算アルゴリズムは全く異なる.
  - **subtype polymorphism**(**部分型多相**): 操作の対象の複数の「型」の範囲が，型上の順序 (部分型関係) に基づいて決定される． (例) Java 言語におけるフィールドアクセス.
  - **parametric polymorphism**(**パラメタ多相**): 操作の対象の「型」に関わらず，同じ操作を行なう場合に発生する多相性．(例) 数学において，$\circ$ は，g の値域と f の定義域さえ等しければ，**任意の**定義域・値域をもつ 関数 f, g の関数合成 $f \circ g$ を表す．また，ソートアルゴリズムは (要素の比較に用いるアルゴリズムを除いて) その要素の型には依存しないため，一般にパラメタ多相性をもつ.

オブジェクト指向言語において polymorphism というと,「あるメソッド呼び出し式において呼び出されるメソッドが，呼び出されるオブジェクトの (静的型ではなく) 実行時の型

(クラス) によって決まること」を指すことが多い．これは ad-hoc polymorphism と subtype polymorphism の組み合わせであると考えられる．

ここでは，λ 計算上にパラメタ多相性を表現した，System F を紹介する．

2 System F の概要

2.1 型抽象と型適用

単純型付 λ 計算による，int 上の関数を合成する関数の定義

$$\begin{aligned}&\texttt{compose} \in (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int}\\ &= \lambda\texttt{f}{\in}\texttt{int} \to \texttt{int}.\lambda\texttt{g}{\in}\texttt{int} \to \texttt{int}.\lambda\texttt{x}{\in}\texttt{int}.\texttt{f (g x)}\end{aligned}$$

とその使用例

$$\texttt{compose } (\lambda\texttt{x}{\in}\texttt{int}.\texttt{x+1}) \ (\lambda\texttt{y}{\in}\texttt{int}.\texttt{y+2})$$

に注目してみる．よく考えてみると，任意の型 T_1, T_2, T_3 に対し，

$$\begin{aligned}&\texttt{compose} \in (T_1 \to T_2) \to (T_3 \to T_1) \to T_3 \to T_2\\ &= \lambda\texttt{f}{\in}T_1 \to T_2.\lambda\texttt{g}{\in}T_3 \to T_1.\lambda\texttt{x}{\in}T_3.\texttt{f (g x)}\end{aligned}$$

という定義が受理される．この compose (の右辺) は，3 つの型を与えると，その型に応じた関数合成操作を返す「型から項への関数」と考えることができる．System F では，このような，**多相関数**(*polymorphic function*) と呼ばれる関数を体系の内部の概念として捉えることが可能である．

上の T_i は型を表すメタ変数だが，このような型からの関数を構成するためには，型のパラメータを表現する記法が必要である．このようなパラメータとして，**型変数**(*type variable*) を導入する．型変数を表す記号 (メタ変数) として X, Y などを使用する．型変数は int などと同様に通常の型として (その有効範囲内で) 使うことができる．

- **型抽象**(*type abstraction*): $\lambda X.t$ という形で型 X を受け取って t を返す多相的関数を表現．$\lambda X.$ は型変数 X の宣言であり，その有効範囲は t である．t の型が T である時，型抽象 $\lambda X.t$ の型は，$\forall X.T$ と表現される．このような型を**全称型**(*univesal type*) と呼ぶ．$\forall$ は論理で使われているように,「任意の X について…」と読むことができる．上の compose を多相的関数として定義すると，

  $$\begin{aligned}&\texttt{compose} \in \forall X.\forall Y.\forall Z.(X \to Y) \to (Z \to X) \to Z \to Y\\ &= \lambda X.\lambda Y.\lambda Z.\lambda\texttt{f}{\in}X \to Y.\lambda\texttt{g}{\in}Z \to X.\lambda\texttt{x}{\in}Z.\texttt{f (g x)}\end{aligned}$$

  となる．$\lambda\texttt{f}{\in}X \to Y.\cdots$ の部分の型は，単に X, Y, Z を int などと同じ型だと思えば，単純型付 λ 計算と同じ理屈で型を求めることができる．

- **型適用**(*type application*): t[T] という形で t という polymorphic function を型の実引数 T に適用している． t の型が $\forall X.T_0$ である時，全体の型は T に関して具体化・特化した型，T_0 中の X を T で置き換えたような型になる．例えば上で定義した多相的な compose の定義を使って，int 上の関数の合成をする場合は，

  $$\texttt{compose [int][int][int] (}\lambda\texttt{x}\in\texttt{int.x+1)(}\lambda\texttt{y}\in\texttt{int.y+2)}$$

  と記述することになる．この部分項 compose [int][int][int] の型が，

  $$(\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int}$$

  となる．

2.2 Predicative/impredicative polymorphism

数学・計算機科学における「変数」という概念には，通常，その値域 (動く範囲) が (少なくとも暗黙に) 定められている．型変数の動く範囲として，(従来からの型である) 単純型のみである場合，そのパラメタ多相型システムは **predicative**(**叙述的**) である，といい，$\forall X.T$ のような型をも含む場合，**impredicative**(**非叙述的**) である，という．System F の多相型システムは非叙述的である．非叙述的な型システムは，全称型の意味を決定する型変数の動く範囲に，これから定義しようとする型全体が含まれる，という意味で循環性があり，ある意味奇妙な型システムである．この結果，

$$\texttt{id} \in \forall X.X \to X = \lambda X.\lambda x{\in}X.x$$

という恒等関数 id に対し，型引数として，id **それ自身の型**$\forall X.X \to X$ を与え，自己適用のようなことができる．

$$\texttt{id [}\forall X.X \to X\texttt{] id}$$

id [$\forall X.X \to X$] の部分の型は $(\forall X.X \to X) \to (\forall X.X \to X)$ となっている．

3 プログラミング in System F

3.1 型付 Church Boolean

真偽値の型は，$\forall X.X \to X \to X$ と表現される．

$$\begin{aligned} \texttt{true} &= \lambda X.\lambda\texttt{x}{\in}X.\lambda\texttt{y}{\in}X.\texttt{x} \\ \texttt{false} &= \lambda X.\lambda\texttt{x}{\in}X.\lambda\texttt{y}{\in}X.\texttt{y} \\ \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 &\stackrel{\text{def}}{=} t_1[T]\ t_2\ t_3 \end{aligned}$$

ただし T は t_2, t_3 の型である．

3.2 Pairing

t_1, t_2 の型を T_1, T_2 とすると，その組の型は $\forall X.(T_1 \to T_2 \to X) \to X$ と表現される．

$$\begin{aligned}
\texttt{pair} &= \lambda X.\lambda Y.\lambda \texttt{x}{\in}X.\lambda \texttt{y}{\in}Y.\lambda Z.\lambda \texttt{f}{\in}X \to Y \to Z.\texttt{f x y} \\
\texttt{fst} &= \lambda X.\lambda Y.\lambda \texttt{p}{\in}\forall Z.(X \to Y \to Z) \to Z.\texttt{p [}X\texttt{]}\ (\lambda \texttt{f}{\in}X.\lambda \texttt{s}{\in}Y.\texttt{f}) \\
\texttt{snd} &= \lambda X.\lambda Y.\lambda \texttt{p}{\in}\forall Z.(X \to Y \to Z) \to Z.\texttt{p [}Y\texttt{]}\ (\lambda \texttt{f}{\in}X.\lambda \texttt{s}{\in}Y.\texttt{s})
\end{aligned}$$

以下で $\texttt{pair [}T_1\texttt{] [}T_2\texttt{]}\ t_1\ t_2$ (T_1, T_2 は t_1, t_2 の型) は $\langle t_1, t_2 \rangle$ と書く．

3.3 型付 Church 数

Church 数の型は $\forall X.(X \to X) \to X \to X$ と表現される．以下で，この型を Nat と書く．

$$\begin{aligned}
\texttt{0} &\stackrel{\text{def}}{=} \lambda X.\lambda \texttt{s}{\in}X \to X.\lambda \texttt{z}{\in}X.\texttt{z} \\
n &\stackrel{\text{def}}{=} \lambda X.\lambda \texttt{s}{\in}X \to X.\lambda \texttt{z}{\in}X.\underbrace{\texttt{s}\ (\texttt{s}\ (\cdots\ \texttt{s}}_{n}(\texttt{z})\cdots)) \\
\texttt{succ} &= \lambda \texttt{n}{\in}\texttt{Nat}.\lambda X.\lambda \texttt{s}{\in}X \to X.\lambda \texttt{z}{\in}X.\texttt{s}\ (\texttt{n [}X\texttt{]\ s z}) \\
\texttt{plus} &= \lambda \texttt{n}{\in}\texttt{Nat}.\lambda \texttt{m}{\in}\texttt{Nat}.\lambda X.\lambda \texttt{s}{\in}X \to X.\lambda \texttt{z}{\in}X.\texttt{n [}X\texttt{]\ s}\ (\texttt{m [}X\texttt{]\ s z}) \\
\texttt{times} &= \lambda \texttt{n}{\in}\texttt{Nat}.\lambda \texttt{m}{\in}\texttt{Nat}.\lambda X.\lambda \texttt{s}{\in}X \to X.\lambda \texttt{z}{\in}X.\texttt{n [}X\texttt{]}\ (\texttt{m [}X\texttt{]\ s})\ \texttt{z} \\
\texttt{pred} &= \lambda \texttt{n}{\in}\texttt{Nat}. \\
&\quad \texttt{fst [Nat] [Nat]} \\
&\quad\ (\texttt{n [}\forall Z.(\texttt{Nat} \to \texttt{Nat} \to Z) \to Z\texttt{]} \\
&\quad\quad (\lambda \texttt{p}{\in}\forall Z.(\texttt{Nat} \to \texttt{Nat} \to Z) \to Z. \\
&\quad\quad\quad \langle \texttt{snd [Nat] [Nat] p}, \texttt{succ}\ (\texttt{snd [Nat] [Nat] p}) \rangle) \\
&\quad\quad (\langle \texttt{0}, \texttt{0} \rangle))
\end{aligned}$$

型抽象・型適用を無視すれば，型無しバージョンが得られることに注意．pred の定義では型適用の引数として $\forall$... という形をしている Nat が渡されており impredicativity が利用されている．

4 形式的定義

4.1 型の定義

本来の System F は，単純型付 λ 計算に，型変数・型抽象・型適用を加えたような体系であるが，ここでは型変数の導入に伴なって， 環境に $X = T$ のような宣言を許すことによって，型の定義機構も導入する． $X = T$ という宣言はその有効範囲内で X が T と同一であ

$$\frac{\Gamma \in \mathbf{Env}}{\Gamma, X \in \mathbf{Env}} \quad (\textsc{Env-TDecl}) \qquad \frac{\Gamma \in \mathbf{Env} \qquad \Gamma \vdash t \in T}{\Gamma, x \in T = t \vdash \#^0 x \in T} \quad (\textsc{T-Var1})$$

$$\frac{\Gamma \vdash T \in \mathbf{Type}}{\Gamma, X = T \in \mathbf{Env}} \quad (\textsc{Env-TDefn}) \qquad \frac{\Gamma, x \in T_1 \vdash t \in T_2}{\Gamma \vdash \lambda x \in T_1.t \in T_1 \to T_2} \quad (\textsc{T-Abs})$$

$$\frac{\Gamma \in \mathbf{Env}}{\Gamma, X \vdash X \in \mathbf{Type}} \quad (\textsc{Ty-TVar0}) \qquad \frac{\Gamma \vdash t_1 \in T_1 \to T_2 \qquad \Gamma \vdash t_2 \in T_1}{\Gamma \vdash t_1\ t_2 \in T_2} \quad (\textsc{T-App})$$

$$\frac{\Gamma \vdash T \in \mathbf{Type}}{\Gamma, X = T \vdash X \in \mathbf{Type}} \quad (\textsc{Ty-TVar1}) \qquad \frac{\Gamma, X \vdash t \in T}{\Gamma \vdash \lambda X.t \in \forall X.T} \quad (\textsc{T-TAbs})$$

$$\frac{\Gamma \vdash T_1 \in \mathbf{Type} \qquad \Gamma \vdash T_2 \in \mathbf{Type}}{\Gamma \vdash T_1 \to T_2 \in \mathbf{Type}} \quad (\textsc{Ty-Arr}) \qquad \frac{\Gamma \vdash t \in \forall X.T_1 \qquad \Gamma \vdash T_2 \in \mathbf{Type} \qquad \Gamma, X = T_2 \vdash T_1[X] \Rightarrow T'}{\Gamma \vdash t[T_2] \in T' \downarrow_X} \quad (\textsc{T-TApp})$$

$$\frac{\Gamma, X \vdash T \in \mathbf{Type}}{\Gamma \vdash \forall X.T \in \mathbf{Type}} \quad (\textsc{Ty-All})$$

$$\frac{\Gamma \in \mathbf{Env}}{\Gamma, x \in T \vdash \#^0 x \in T} \quad (\textsc{T-Var0}) \qquad \frac{\Gamma \vdash t \in T_1 \qquad \Gamma \vdash T_1 = T_2}{\Gamma \vdash t \in T_2} \quad (\textsc{T-Eq})$$

図 1: 多相型付 λ 項の定義 (抜粋)

るという意味だが，これにより字面上等しくない型が実質的には等しいという状態が発生する．例えば，

```
X = int->int, f : X
```

という宣言のもとで，`f 1` は `int` 型の項になるが，`f` の宣言に付加された型は関数型ではない．このため，(定義の詳細は省略するが),「ふたつの型が等しい」という判断 $\Gamma \vdash T_1 = T_2$ を導入し，項の型は等しい型で置き換えることができる，という規則が必要になる．また，型検査アルゴリズムも少々複雑になる．

4.1.1 定義: 判断 $\Gamma \in \mathbf{Env}$ および，$\Gamma \vdash T \in \mathbf{Type}$，型判断 $\Gamma \vdash t \in T$ の定義 (抜粋) を図 1 に示す．

4.1.2 定義: 簡約関係の判断 $\Gamma \vdash t \longrightarrow t'$ は以下の規則で定義される．($\Gamma \vdash t[\#^i x] \Rightarrow t'$ の定義は省略している．) □

5 型検査アルゴリズム

型検査のアルゴリズムは単純型付き λ 計算のものを拡張することで得られる．型の等価性を考慮しなければならないので，例えば関数適用項の型で T_1 が関数型でなくても，型変数

$$\frac{}{\Gamma, x \in T = t \vdash x \longrightarrow t\uparrow_x} \quad \text{(E-Def)}$$

$$\frac{\Gamma \vdash \#^i x \longrightarrow t}{\Gamma, y \in T \vdash \#^i x\uparrow_y \longrightarrow t\uparrow_y} \quad \text{(E-Shift1)}$$

$$\frac{\Gamma \vdash \#^i x \longrightarrow t}{\Gamma, y \in T' = t' \vdash \#^i x\uparrow_y \longrightarrow t\uparrow_y} \quad \text{(E-Shift2)}$$

$$\frac{\Gamma, x \in T_2 = t_2 \vdash t_1[\#^0 x] \Rightarrow t'}{\Gamma \vdash (\lambda x \in T_2.t_1)\ t_2 \longrightarrow t'\downarrow_x} \quad \text{(E-Beta)}$$

$$\frac{\Gamma, X = T \vdash t_0[\#^0 X] \Rightarrow t'}{\Gamma \vdash (\lambda X.t_0)\,[T] \longrightarrow t'\downarrow_X} \quad \text{(E-TBeta)}$$

$$\frac{\Gamma \vdash t_1 \longrightarrow t_1'}{\Gamma \vdash t_1\,[T] \longrightarrow t_1'\,[T]} \quad \text{(E-TApp)}$$

$$\frac{\Gamma, X \vdash t \longrightarrow t'}{\Gamma \vdash \lambda X.t \longrightarrow \lambda X.t'} \quad \text{(E-TAbs)}$$

図 2: 多相型付 λ 計算: 簡約関係

である場合には，関数型として定義されている可能性があるので，定義を展開する必要がある．環境 Γ，項 (らしきもの)t を入力として，$\Gamma \vdash t \in T$ なる T (あれば) を返す，アルゴリズム $TC(\Gamma, t)$ を，型定義の展開を行う補助関数 *Expand* とともに以下に示す.

$$\begin{array}{ll}
\mathit{Expand}((\Gamma, X = T), \#^0 X) = & \textbf{if } T \text{ is a type variable } Y \textbf{ then } \mathit{Expand}(\Gamma, Y) \textbf{ else } T \\
\mathit{Expand}((\Gamma, X = T), \#^{i+1} X) = & \mathit{Expand}(\Gamma, \#^i X) \\
\quad \vdots & \\
TC((\Gamma, x \in T), \#^0 x) = & \textbf{if } TC_{env}(\Gamma) \textbf{ then } T \textbf{ else } \mathit{fail} \\
TC((\Gamma, x \in T = t), \#^0 x) = & \textbf{if } TC_{env}(\Gamma) \textbf{ then } T \textbf{ else } \mathit{fail} \\
TC((\Gamma, x \in T'), \#^{i+1} x) = & TC(\Gamma, \#^i x) \\
\quad \vdots & \\
TC(\Gamma, \lambda x{\in}T_1.t) = & \textbf{let } T_2 = TC((\Gamma, x \in T_1), t) \textbf{ in } T_1 \to T_2 \\
TC(\Gamma, t_1\ t_2) = & \textbf{let } T_1 = TC(\Gamma, t_1) \textbf{ in} \\
 & \textbf{let } T_2 = TC(\Gamma, t_2) \textbf{ in} \\
 & \textbf{if there exists } T_3 \text{ s.t. } \mathit{Expand}(\Gamma, T_1) = T_2 \to T_3 \textbf{ then } T_3 \textbf{ else } \mathit{fail} \\
TC(\Gamma, \lambda X.t_0) = & \textbf{let } T_0 = TC((\Gamma, X), t_0) \textbf{ in } \forall X.T_0 \\
TC(\Gamma, t_1\,[T]) = & \textbf{let } T_1 = TC(\Gamma, t_1) \textbf{ in} \\
 & \textbf{if there exists } T_1' \text{ s.t. } \mathit{Expand}(\Gamma, T_1) = \forall X.T_1' \textbf{ then } [T/X]T_1' \textbf{ else } \mathit{fail}
\end{array}$$

6 諸性質

6.1 定理 [Type Preservation]: $\Gamma \vdash t \in T$ かつ $\Gamma \vdash t \longrightarrow t'$ ならば，$\Gamma \vdash t' \in T$ である.

System F は Church 数の指数関数などを表現できる程，表現力が豊かであるにも関らず，全てのプログラムの実行は停止する．(つまりチューリング完全ではない．)

6.2 定理 [Strong Normalization]: $\Gamma \vdash t \in T$ かつ Γ, t が `fix` を含まないならば $\Gamma \vdash t \longrightarrow t_1 \longrightarrow \cdots \longrightarrow t_n \longrightarrow \cdots$ なる無限列は存在しない.