ソフトウェア基礎論配布資料 (3)
λ計算 (1)

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 16 年 10 月 19 日

1 関数とλ記法，λ計算

プログラムにおいて，似たような式・計算手順が複数の箇所で必要になった時には，関数や手続きを定義して，式・手順の再利用を図ることが一般的である．数学でも

$$5^2\pi + 7^2\pi + 20^2\pi$$

と書く代わりに，

$$f(2) + f(7) + f(20) \quad \text{ただし } f(x) = x^2\pi$$

と書けば，式の見通しがよくなる．ここで x はパラメータと呼ばれ，使われる場所によって異なる部分を表す役割を担っている．

この f のように，関数の概念は，集合論で扱われる「入力と出力の対の集合」という扱いよりも，「入力から出力を計算する式」とする扱いの方がより直感的かもしれない．このような「計算可能な関数」を扱うための理論が **λ計算**(*λ-calculus*) であり，その中心となるのが λ 記法と呼ばれる関数の記法である．

λ 記法は，λ⟨ パラメータ ⟩.⟨ 式 ⟩ という形で「パラメータを入力として式の計算結果を出力とする関数」を表現する．上の例の f は $\lambda x.x^2\pi$ と書くことができる．この λ 記法の特徴的な点は，

その関数自体に名前をつけずに関数を表現することができる

という点である．これにより，関数の概念そのものと関数に名前をつけるという行為を切離すことができる．

λ 記法による関数に，その引数を与えた時 (関数を適用する，という) 関数の値は「パラメータを入力で置き換えること」で得ることができる．すなわち，$f = \lambda x.x^2\pi$ とすると，

$$\begin{aligned} & f(2) + f(7) + f(20) \\ (\text{定義より}) \quad &= (\lambda x.x^2\pi)(2) + f(7) + f(20) \\ (x \text{ を } 2 \text{ で置き換え}) \quad &= 2^2\pi + f(7) + f(20) \\ &\vdots \\ &= 453\pi \end{aligned}$$

という推論 (計算) が可能になる．λ 計算の体系は，λ 記法による関数，関数適用によるパラメータ置換の仕組みを形式的に実現したものであり，特に，パラメータ置換こそが計算ステップである，という立場をとる．

1.1 なぜ λ 計算なのか？

- 高級プログラミング言語にとって，関数・手続きの仕組みは必須のものであり，λ 計算は関数呼び出しの仕組みのモデルである．
- 様々なプログラム機構が λ 計算の枠組で「表現できる」．
- チューリング機械より単純，かつプログラミング言語に近い計算モデル．

2 λ計算概観

ここでは，厳密な定義に入る前に，形式化の概観をみていき，算術式の言語にはなかった概念を導入する．

おおまかな文法 上の例では，π という実数や乗算を仮定していたが，純粋な λ 計算においては，通常のプログラミング言語に見られるような整数などの基本的なデータ (と，その上の演算) すら取り扱わない，関数と関数適用しかない言語になっている．純粋な体系上に，基本的なデータ型を付加することは容易であるので，後で扱うことにする．

前回の体系における算術式のように，λ 計算における計算の対象を**項**(*term*) と呼ぶ．term の文法はメタ変数を t として次のように与えられる．

$$t ::= x \mid \lambda x.t \mid t\ t$$

x は変数 (項) と呼ばれ関数パラメータを示す．$\lambda x.t$ は λ **抽象**(*lambda abstraction*) と呼ばれ，直感的には x をパラメータとして t を計算する関数を示す．$t_1\ t_2$ は**関数適用**(*function application*) と呼ばれ，関数 t_1 を実引数 t_2 に適用する計算を示す．

変数項として使えるものの集合を **Var** と呼び，少なくとも `x`, `y`, `z`, `x'`, `y2` といった文字列を含むとする．上の項の定義における x は変数項を表すメタ変数である．同じアルファベットを使っているが意味は違うものなので注意する必要がある．

宣言の有効範囲と変数束縛，自由変数 プログラム中で変数を使用するときには，通常，使用する旨の**宣言**(*declaration*) を伴なう．また，宣言された変数には，使ってもよい場所があり，例えば，関数/手続きのパラメータとして宣言された変数は，その関数/手続き内でしか使えない．このような「使ってもよい場所」を (変数) 宣言の**有効範囲**(*scope*) と呼ぶ．逆に，変数を使用している箇所は何らかの宣言の有効範囲下にあり，**束縛変数**(*bound variable*) と呼ばれる．

λ 計算において，$\lambda x.t$ の x が変数宣言であり，その有効範囲は t である．t に現われる変数項 x は束縛変数である．束縛変数ではない変数を**自由変数**(*free variable*) と呼ぶ．例えば，項 `λx.λy.x z` において，`x` は束縛変数であり，`z` は自由変数である．項 `(λx.x) (λx.x)` や項 `λx.(λx.x)` のように，同じ変数が二箇所以上で宣言されていることも考えられる．この場合，変数項がどの宣言を参照して

いるかを意識することが重要である．(後者の場合，有効範囲が入れ子状に重なっているが，変数項 x は内側の宣言を参照する．) また，項 (λx.x) x では，x は自由変数/束縛変数の両方として出現していることになる．これらのことから，自由・束縛という概念は，より正確には，変数 (の名前) ではなく，変数の出現 (箇所) に対して使われると考えた方がよい．

このような用語を導入したのは，関数のパラメータとは何かをはっきりさせるためであり，「パラメータを引数で置き換える」と説明した関数適用の動作をより正確に記述するためである．つまり，関数適用の動作は，より正確には「$(\lambda x.t_1)\ t_2$ は，この x の有効範囲内で束縛されている x の出現のみを t_2 で置き換える」ということになる．例えば，

$$\mathtt{x}\ ((\lambda \mathtt{x}.(\mathtt{x}\ (\lambda \mathtt{x}.\mathtt{x})))\ \mathtt{y})$$

という項は，

$$\mathtt{x}\ (\mathtt{y}\ (\lambda \mathtt{x}.\mathtt{x}))$$

になるのであって，

$$\mathtt{x}\ (\mathtt{y}\ (\lambda \mathtt{x}.\mathtt{y}))$$

や

$$\mathtt{y}\ (\mathtt{y}\ (\lambda \mathtt{x}.\mathtt{x}))$$

にはならないのである．

簡約戦略・値呼び・名前呼び　関数適用によるパラメータ置換の過程を β **簡約**(β- *reduction*) と呼ぶ．また，β 簡約が可能な形の項を β **基**(β-*redex*) と呼ぶ．また，後で定義するように「部分項を簡約する」という term の 2 項関係を簡約関係と呼び，算術式の言語にならって $\longrightarrow$ という記号 (に添字をつけて) で表現する．

一般に，項の中には β 基が複数ある可能性があるが，どの β 基から簡約していくかについては，算術式の言語でみたように，任意の順序が可能 (簡約は非決定的) な定義と適当な順序を定める定義が考えられ，簡約関係にも何通りか考えることができる．通常，(逐次の) プログラミング言語では，計算順序は決定的に定まっているが，簡約が決定的になるように簡約関係を定めることを，**簡約戦略**(*reduction strategy*) を定めるという．

以下，代表的な簡約関係を紹介する．

full β-reduction: $\longrightarrow_f$　full β-reduction は，β 基なら，どこでも β 簡約してよいという関係である．どこからでも簡約してよいので複数の計算過程を持つ可能性がある．

normal order reduction: $\longrightarrow_n$　最外最左簡約ともよばれる簡約戦略である．最も外側，最も左側にある β 基から優先して簡約していく．後で見るように，λ 計算における簡約は有限ステップで正規形に到達せず，計算が終了しない場合があるが，normal order reduction に関しては，「full β-reduction で正規形 (β 基がひとつもない項) に到達する簡約列がひとつあるなら，normal order reduction でも必ず正規形に到達する」という性質がある．

通常のプログラミング言語では，関数/手続き本体の計算はパラメータが与えられるまで発生しないが，上記ふたつの簡約は，λ 抽象の中も簡約が許されているという点で，それとは異なっている．以下のふたつの簡約戦略は「λ の中は簡約しない」戦略である．

call-by-value reduction: $\longrightarrow_v$　多くのプログラミング言語では関数/手続き呼び出しは，引数の計算が終わってから行なわれる．このような呼び出し方法を**値呼び**(*call by value*) と呼ぶ．これをモデル化したのが call-by-value reduction である．

call-by-name reduction　一方，Algol, Miranda, Haskell といった言語では，引数の計算をする前に関数呼び出しが発生する．引数は関数本体内で必要になった時にのみ，その値が計算される．このような呼び出し方式を **名前呼び**(*call by name*) という．(実際の Haskell では，名前呼びを改良した call by need という方式が用いられている．)

3　プログラミング言語: λ 計算

純粋な λ 計算をプログラミング言語のモデルとして捉えるには,

- 関数呼び出しのモデルといいつつ，引数がふたつ以上ある関数すらない.
- 変数は全て関数パラメータで局所変数の宣言すらできない.
- 再帰的関数が考えられていない
- 基本的なデータ型すら入っていない

など一見すると余りにも貧弱にみえる．しかし，これらは実は全て λ 計算の中で表現できることである．以下では，様々なプログラミング要素が純粋な λ 計算の枠組みで表現できることを見る.

3.1　複数の引数をもつ関数と Curry 化

$$x \text{ と } y \text{ をパラメータとして } t \text{ を計算する関数}$$
$$= x \text{ をパラメータとして,「} y \text{ をパラメータとして } t \text{ を計算する関数」を返す関数}$$

3.2　定義

局所変数 x を宣言して，その初期値を t_1 として，t_2 を実行する，ことを `var` x `=` t_1`;` t_2 と記述する.

$$\texttt{var } x \texttt{ = } t_1\texttt{; } t_2 \Longrightarrow (\lambda x.t_2)\, t_1$$

3.3　Church Boolean

$$\begin{aligned} \texttt{true} &\Longrightarrow \lambda\texttt{x.}\lambda\texttt{y.x} \\ \texttt{false} &\Longrightarrow \lambda\texttt{x.}\lambda\texttt{y.y} \\ \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 &\Longrightarrow (t_1\ t_2)\, t_3 \end{aligned}$$

3.4 Pairing

$$\begin{aligned} \texttt{pair}\ t_1\ t_2 &\Longrightarrow \lambda\texttt{x.x}\ t_1\ t_2 \\ \texttt{fst} &\Longrightarrow \lambda\texttt{p.p}\ (\lambda\texttt{f.}\lambda\texttt{s.f}) \\ \texttt{snd} &\Longrightarrow \lambda\texttt{p.p}\ (\lambda\texttt{f.}\lambda\texttt{s.s}) \end{aligned}$$

3.5 Church 数

$$\begin{aligned} \texttt{0} &\Longrightarrow \lambda\texttt{s.}\lambda\texttt{z.z} \\ n &\Longrightarrow \lambda\texttt{s.}\lambda\texttt{z.}\underbrace{\texttt{s}\ (\texttt{s}\ (\cdots\ \texttt{s}}_{n}(\texttt{z})\cdots)) \\ \texttt{plus} &\Longrightarrow \lambda\texttt{n.}\lambda\texttt{m.}\lambda\texttt{s.}\lambda\texttt{z.n s (m s z)} \\ \texttt{times} &\Longrightarrow \lambda\texttt{n.}\lambda\texttt{m.}\lambda\texttt{s.}\lambda\texttt{z.n (m s) z} \end{aligned}$$

3.6 再帰的関数定義

$$\begin{aligned} \texttt{omega} &\stackrel{def}{\equiv} (\lambda\texttt{x.x x})\ (\lambda\texttt{x.x x}) \\ \texttt{fix} &\stackrel{def}{\equiv} \lambda\texttt{f.}((\lambda\texttt{x.f}\ (\texttt{x x}))\ (\lambda\texttt{x.f}\ (\texttt{x x}))) \\ \texttt{fixv} &\stackrel{def}{\equiv} \lambda\texttt{f.}((\lambda\texttt{x.f}\ (\lambda\texttt{y.(x x)}\ \texttt{y}))\ (\lambda\texttt{x.f}\ (\lambda\texttt{y.(x x)}\ \texttt{y}))) \end{aligned}$$

$f(x) = e$ (式 e には f が出現) なる再帰的関数は $\texttt{fix}\ (\lambda\texttt{f.}\lambda\texttt{x.e})$ と表現される.