第 2 回

Rei Tomori

5/15/2025.

5. 型無しラムダ計算

5.1. 基礎

5.1. 関数による抽象化

- プログラム中に繰返し出現する計算を，手続き (関数) により抽象化することを考えよう．
  - 関数は一つ以上の名前付き引数を取り， 引数に値を渡すことで具体化される．
    - e.g.) 式 $(5*4*3*2*1) + (7*6*5*4*3*2*1) - (3*2*1)$ は関数 **factorial** により，

$$\mathbf{factorial}(5) + \mathbf{factorial}(7) - \mathbf{factorial}(3) \tag{1}$$

      のようにかける．
    - ただし，各 $n > 0$ に対し，

$$\mathbf{factorial}(n) ::= \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n\ *\ \mathbf{factorial}(n-1) \tag{2}$$

5.1. 関数による抽象化 (cont'd)

- 関数による抽象化
  - 関数が各 n に対して値を返すことを $\lambda n.\ ...$ と書くことにすると，**factorial** は

$$\mathbf{factorial} = \lambda n.\mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * \mathbf{factorial}(n-1) \tag{3}$$

    のように定義し直せる．
    - すると，$\mathbf{factorial}(0)$ は，**factorial** の本体 $\mathbf{if}\ n = 0 ...$ の引数の変数 n を 0 で置き換えて評価した結果として得られる値を指す．(i.e. 1)

5.1. 関数による抽象化 (cont'd)

- 関数による抽象化
  - 関数が各 n に対して値を返すことを $\lambda n.\ \ldots$ と書くことにすると，**factorial** は

$$\mathbf{factorial} = \lambda n.\mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * \mathbf{factorial}(n-1) \tag{4}$$

    のように定義し直せる．
    - すると，**factorial**(0) は，**factorial** の本体 **if** $n = 0 \ldots$ の引数の変数 n を 0 で置き換えて評価した結果として得られる値を指す．(i.e. 1)

以上より，関数の引数指定と関数適用をもつ言語で計算を表現しうることがわかった．そこで，これらを備えた小さな言語である **(型無し)λ 計算**を導入し，その性質を考える．

5.1. 型無し λ 計算の構文

λ 計算の構文を定義しよう.

定義 (λ 計算の構文)

λ 計算の構文を次に定める. ただし x は変数を, t は項をそれぞれ表わすメタ変数とする.

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

構文に括弧が含まれていないのは, この定義が**抽象構文木**を定めていることに因る.

5.1. 抽象構文と具象構文

プログラミング言語には**具象構文**と**抽象構文**の 2 つの構文的レベルがある．*1

- 具象構文 : 具体的な文字列．生成文法で定義
- 抽象構文 : より単純な内部表現．抽象構文木 (AST) で表現

具象構文から AST への変換は，字句解析器によるトークン化を経て構文解析器でパースされることで行なわれる．

- 演算子の優先順位や結合規則を適切に定めることで，プログラムで括弧を多用せずとも AST を一意に定められる．
  - e.g.) 式 $1 + 2*3$ は一意に $1 + (2*3)$ のように解釈される．

1 他にも，言語を核となる機能だけからなる制限された**中核言語**と，中核言語の機能から構成される派生形式を含む完全な**外部言語**に分けることもあるが，ここでは説明を割愛する．

5.1. 抽象構文と具象構文 (cont'd)

注意 (関数適用と抽象の結合性)

関数適用は左結合，抽象は右結合とする．

したがって， たとえば式 $\lambda x.\lambda y.x\ y\ x$ の AST は $\lambda x.(\lambda y.(x\ y)\ x)$ と同じである．

5.1. 変数とメタ変数

注意 (メタ変数の使用について)

メタ変数 t および s, u を (添字付きおよび添字無しで) は任意の項を，メタ変数 x(および y, z) は任意の変数を表わすとする．

メタ言語と対象言語において同じ変数を使う場合は文脈から判断する．

- 例えば，「項 $\lambda x.\lambda y.x\ y$ は $z = x$ と $s = \lambda y.x\ y$ の下で $\lambda z.y$ の形」という場合には，x, y は対象言語の変数，z, s はメタ変数．

5.1. スコープ

定義 (自由出現・束縛出現)

抽象 $\lambda x.t$ の本体 t の中に x が出現するとき，その x の出現は**束縛**されているといい，そうでない場合は**自由**であるという．

たとえば，$(\lambda x.x)x$ において，1 つめの出現は束縛されており，2 つめの出現は束縛されていない，つまり自由出現である．

定義 (閉じた項)

自由変数のない項は**閉じている**といい，これらの項 (閉じた項) を**コンビネータ**という．

5.1. 操作的意味論

純粋な，i.e. 組込みの定数や演算子を持たない λ 計算において，"計算"とは引数の関数への適用である．

- i.e. 左側が λ 抽象であるような関数適用 (**簡約基** ;redex) を，抽象本体の束縛変数の各自由出現を右側の項で置き換えた項に書き換える：

$$(\lambda x.t_{12})t_2 \to [x \mapsto t_2]t_{12} \tag{5}$$

この書き換え操作を**ベータ簡約**という．各ステップで評価できる redex は，評価戦略によって定められる．

5.1. 操作的意味論 (2)

定義 (完全 β 簡約)

任意の簡約基がいつでも簡約できるような評価戦略を**完全ベータ簡約**という．

定義 (正規順序戦略)

最左・最外簡約基が最初に簡約されるような評価戦略を**正規順序戦略**という．

条件を満たす簡約基は高々一つなので，各項 t は高々一つの項 t' に簡約される．

5.1. 操作的意味論 (cont'd)

正規順序戦略を強め，抽象内部の簡約を禁じよう．*2

定義 (名前呼び戦略)

最左・最外簡約基から最初に簡約し，抽象内部での簡約を許さない評価戦略を**名前呼び戦略**という．

例 (call-by-name の簡約列の例)

次は名前呼びで項 $\mathbf{id}(\mathbf{id}(\lambda z.\mathbf{id}\ z)$ を評価したときの簡約列である．ただし $\mathbf{id} = \lambda z.z$.

$$\mathbf{id}(\mathbf{id}(\mathbf{id}(\lambda z.\mathbf{id}\ z))) \to \mathbf{id}(\lambda z.\mathbf{id}\ z) \to \lambda z.\mathbf{id}\ z$$

2 Haskell などでは必要呼び，つまり一度評価された引数をメモ化する名前呼び戦略を採用している．

5.1. 操作的意味論 (cont'd)

名前呼び戦略では未評価の値を関数に渡す，つまり**非正格** (あるいは遅延評価) である．対照的に，以下の必要呼び戦略では，値は関数に適用される前に必ず評価される (i.e. 正格である)．

定義 (値呼び戦略)

最も外側の簡約基で，右側が値 (i.e. これ以上簡約できない閉じた項) に簡約されている簡約基のみを簡約する戦略を，**必要呼び戦略**という．

例 (call-by-value の簡約列の例)

以下は，値呼びで項 $\mathbf{id}(\mathbf{id})(\lambda z.\mathbf{id}z)$ を評価したときの簡約列である．

$$\mathbf{id}(\mathbf{id}(\lambda z.\mathbf{id}\ z)) \to \mathbf{id}(\lambda z.\mathbf{id}z) \to \lambda z.\mathbf{id}\ z$$

5.2. ラムダ計算でのプログラミング

5.2. 複数の引数

λ 計算で複数の引数を持つ関数を表わす方法を考える．組のコンストラクタと射影を定義してもよいが，高階関数を用いる方が容易である．

例 (カリー化の例)

自由変数 x, y を持つ項 s に対し，全ての引数の組 (v, w) に対して

$$f : (v, w) \mapsto [y \mapsto w][x \mapsto v]s$$

なる関数 f は，$\lambda x.\lambda y.s = \lambda x.(\lambda y.s)$ で与えられる．

5.2.Church ブール値

定義 (ブール値と条件式)

ブール値 $\mathbf{tru}, \mathbf{fls}$(それぞれ $\mathbf{true}$, $\mathbf{false}$ に対応) と条件式 $\mathbf{test}$ は次のように定義される.

$$\begin{aligned} \mathbf{tru} &= \lambda t.\lambda f.t \\ \mathbf{fls} &= \lambda t.\lambda f.f \\ \mathbf{test} &= \lambda l.\lambda m.\lambda n.l\ m\ n \end{aligned}$$

項 $\mathbf{test}\ b\ v\ w \to b\ v\ w$ であり, $\mathbf{tru}(\mathbf{fls})$ が第二引数 (第一引数) を捨てることを用いている.

5.2.Church ブール値 (cont'd)

例 (test 関数の簡約例)

項 $\mathbf{test}\ \mathbf{tru}\ v\ w$ は次のように簡約される：

$$\begin{aligned}\mathbf{test}\ \mathbf{tru}\ v\ w &= (\lambda l.\lambda m.\lambda n.l\ m\ n)\mathbf{tru}\ v\ w \\ &\to(\lambda m.\lambda n.\mathbf{tru}\ m\ n)\to(\lambda n.\mathbf{tru}\ v\ n)w \\ &= \mathbf{tru}\ v\ w\to(\lambda t.\lambda f.t)v\ w\to(\lambda f.v)w\to w\end{aligned}$$

論理積等のブール演算子も関数として定義できる.

- e.g.) 論理積を表わす $\mathbf{and}$ は $\mathbf{and} = \lambda b.\lambda c.b\ c\ \mathbf{fls}$ と書ける. $\mathbf{and}\ b\ c$ の値を第一引数 b について場合分けすれば得られる.

5.2. 二つ組

ブール値を用いることで，二つの値の組 (タプル) を項として表現できる．

定義 (2 つ組)

二つ組のコンストラクタ $\mathbf{pair}$ と射影関数 $\mathbf{fst}, \mathbf{snd}$ を次のように定義する．

$$\mathbf{pair} = \lambda f.\lambda s.\lambda b.b\ f\ s$$
$$\mathbf{fst} = \lambda p.p\ \mathbf{tru}$$
$$\mathbf{snd} = \lambda p.p\ \mathbf{fls}$$

5.2.Church 数

ラムダ項における数 (**Church 数**) は，Peano の自然数同様， 定数 $\mathbf{0}$ と，$\mathbf{0}$ に有限回後者関数を適用した項全体として定義される．

定義 (Church 数)

Church 数 $c_0, c_1, c_2 \ldots$ を，

$$
\begin{aligned}
c_0 &= \lambda s.\lambda z.z \\
c_1 &= \lambda s.\lambda z.s\ z \\
&\vdots \\
c_n &= \lambda s.\lambda z.s^n z
\end{aligned}
$$

と定める．ただし．項 s, z および $n \in \mathbb{N}$ に対して，$s^n z$ は z に s を n 回適用して得られる項とする．

5.2.Church 数 (cont'd)

後者関数 $\mathbf{scc}$ を定める.

定義 (後者関数)

$$\mathbf{scc} = \lambda n.\lambda s.\lambda z.s(n\ s\ z)$$

各 m, n および項 s, z に対して，項 $c_m\ s\ (c_n\ s\ z)$ は $s^m(s^n\ z) = s^{m+n}z$ に評価される．この観察より，Church 数の加算関数 $\mathbf{plus}$ を定義できる.

定義 (加算関数)

$$\mathbf{plus} = \lambda m.\lambda n.\lambda s.\lambda z.m\ s(n\ s\ z).$$

5.2.Church 数 (cont'd)

乗算関数は次のように定義できる.

定義 (乗算関数)

$$\mathbf{times} = \lambda m.\lambda n.m\ (\mathbf{plus}\ n)\ c_0;$$

全く同様に，冪乗関数 **expn** も定義できる.

定義 (冪乗関数)

$$\mathbf{expn} = \lambda m.\lambda n.m\ (\mathbf{times}\ n)\ c_1;$$

あるいは，$\lambda m.\lambda n.m\ n;$.