2007年度
早稲田大学理工学部CS学科
プログラミング言語論

Part 09: λ-calculus

小野 康一 (ONO, Kouichi)
onono@computer.org

このチャートの目的

- λ-calculusについての大まかな知識を得る
- λ-calculusを中心とする計算体系について本気で講義したら一年間くらい必要になるが、ここでは概略だけにとどめる

λ-calculus (lambda calculus)

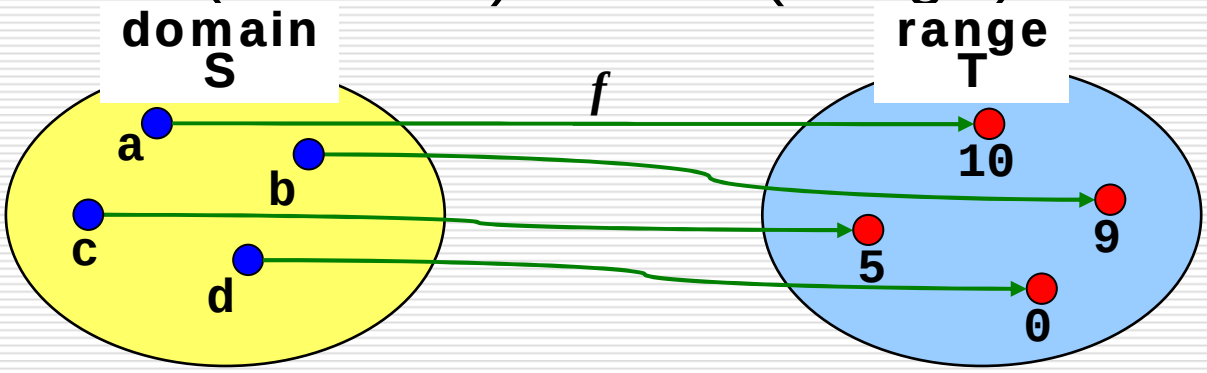
- 和訳：λ計算/ラムダ計算，λ算法/ラムダ算法，など
  - “calculus”は「算術体系/計算体系」のことを意味しているので、単に「計算」と言ってしまうと、少し誤解を招くおそれはある
  - その意味では、「算法」の法が適切かもしれない

λ-calculusとは何なのか

- 実行可能な(計算可能な)計算体系を定義するための理論的な表現形式およびその形式的体系
- Alonzo Churchが考案し、Stephen Cole Kleeneと共に体系化
- すべてを「関数」で表現する

我々が数学などで学ぶ関数

- 集合論的関数
  - 定義域 (domain) と値域 (range) の間の写像



  - 定義域の集合S中のある要素xから値域Tの値への対応を、xへの関数fの適用/作用 (application) によって表す
    f x あるいは *f(x)* と表記

ところで、“f(x)”と書くとき…

- 写像としての関数fそのものを表しているのか(つまり、xは特定の要素ではなく、定義域中の任意の要素を表す変数として書かれている)
- それとも、特定の要素xへの適用を表しているのか
- 文脈によっては不明瞭になる

λ-notation (λ記法)

- xが、関数fの変数である(＝特定の要素ではない)ことを明示するために、xの前に“λ”という記号をおく表記法を考える

  λx f(x)

- あるいは、“x”と“f(x)”の間に“.”を置いて

  λx.f(x)

  のように表記することもある
- この表記法をλ-notation (λ記法) という
- “x”の前に“λ”を置いて、これが関数の変数であると示すことを、λ-binding (λ束縛) という

λ-notationの例

- たとえば、関数f(x)が
  $x^2 + 10$
  である場合、
  $\lambda x\ (x^2 + 10)$ もしくは $\lambda x.(x^2 + 10)$
  と表記する
- プログラミング言語の「関数」を考えると、λ束縛“λx”は仮引数の宣言に相当する(型宣言は除外)

```
int f(int x) {
    return x*x + 10;
}
```

- ここでは型なしλ算法について述べる。型付きλ算法もあるが、ここでは述べない

束縛変数と自由変数

- λ記法で書かれた以下のλ式 (λ-term) があるとする

$$\lambda y\ (x^2 + y)$$

- この式に出現する “y” はλ束縛されている
  - つまり、“y” は束縛変数 (bound variable) である
- 一方で、“x” はλ束縛されていない
  - つまり、“x” は自由変数 (free variable)、もしくは、定義域内の特定の要素を表すシンボルである
- 上記のλ式は、1変数(1引数)の関数を与えている

λ-abstraction (λ抽象)

- 前頁のλ式における自由変数xをλ束縛すると、2変数(2引数)の関数になる

  $\lambda x\ (\lambda y\ (x^2 + y))$

- 簡略化して以下のように表記することにする

  $\lambda x \lambda y\ (x^2 + y)$

- 式Mが与えられている時に、M中で束縛されていないシンボルxをλ束縛して

  $\lambda x\ M$

  なる関数を得る操作を、λ抽象と呼ぶ

適用/作用 (application)

- 関数がλ記法で

  λx M

  と表現される時、λ束縛された変数xに特定の値を代入して、その値からの写像の値を示すことを、関数の適用/作用 (application) と呼び、以下のように表現する (いくつかの表記法がある)

  λx M 3

  λx M (3)　←ここではこの記法を用いる

  (λx M 3)

- 上の表現は関数 λx M の 3 への適用を表している

適用によって得られるモノ

- 関数の適用の結果は、値になるとは限らない
- たとえば、以下の2変数(2引数)の関数があるとする

$$\lambda x \lambda y\ (x^2 + y)$$

- この関数を、1 に適用する

$$\lambda x \lambda y\ (x^2 + y)\ (1)$$

- 適用の結果は以下のようになる

$$\lambda y\ (1 + y)$$

- これは関数になっている

値と関数

- 10とか “abc” という定数の値は、常に同じ適用結果を返す0変数の(引数が0個の)関数だと考えることができる
- もしくは、何に適用しても同じ値を返す関数と考えることができる
- したがって、λ算法において、関数の適用結果が値であっても関数であっても区別しないし、また、λ束縛される変数へ代入されるのが値であっても関数であっても区別しない

2引数以上の関数の表現

- 前頁の関数
  $\lambda x \lambda y\ (x^2+y)$
  は2変数(2引数)の関数
- プログラミング言語で表現すると

```
int f(int x, int y) {
    return x*x+y;
}
```

- しかし、厳密には
  $\lambda x\ (\lambda y\ (x^2+y))$
- これが意味しているのは、どのような関数か?

関数を返す関数

- この関数はどのような関数なのか

  $\lambda x\,(\lambda y\,(x^2+y))$

- xを引数にとって関数を返す関数として定義されており、返された関数は、yを引数にとって値を返す関数になっている
- この関数に3を適用すると

  $\lambda x\,(\lambda y\,(x^2+y))\,(3)$

  その結果として、以下の関数が返される

  $\lambda y\,(9+y)$

  これに5を適用すると14が返される

カリー化と高階関数

- つまり
  $$\lambda x\,(\lambda y\,(x^2+y))\,(3)(5)$$
  の結果は14になる
- 前述のプログラムだとf(3,5)という2引数の関数呼出になるが、λ算法だとこのように1引数の関数の適用の合成で表現する
- このような、2引数以上の関数を1引数の関数の合成で表現する操作を、カリー化 (Currying) と呼ぶ
- カリー化をおこなうと、関数を返す関数や、関数を引数に取る関数が用いられる
- そのような関数を高階関数 (higher order function) と呼ぶ

λ抽象の対象

- 式中のいかなるシンボルもλ束縛可能であり、つまり、どんなシンボルもλ抽象の対象となりうる
- したがって、以下の式

  $x^2 + y$

  がある時、“x” や “y” だけでなく、“+” をλ抽象した関数も定義できる

  $\lambda + (x^2 + y)$
- この関数を “*” という演算子へ適用すると

  $\lambda + (x^2 + y)\ (*)$

  となり、その結果は

  $x^2 * y$

  となる (つまり、関数を関数に適用している)

さて…

- ここまでは、3とか＋といった、算術演算を意識したシンボルを援用して、λ算法を直感的に説明してきた
- しかし、実を言うと、3や＋のような算術演算は、λ算法に必要なものではない
- それらを用いずに、λ記号、変数シンボル、括弧だけで、十分に広い計算を表現できる
- よって、以降では、λx λy ((x (x)) (y))のような、一見なんだかわからないような式が出てくる

厳密なλ式の定義

- 変数それ自身はλ式である
- Mがλ式で、xが変数ならば、

  λx M

  もλ式である
  - λxをこの式の束縛変数部、Mを本体と呼ぶ
- FとAがλ式ならば、AにFを適用した

  F (A)

  もλ式である
  - Fをこの式の演算子部(operator part)、Aを被演算子部(operand part)と呼ぶ

λ式の「計算」

- λ式を「計算する」とは、そのλ式を、等価で、より「単純な」別のλ式へ「変換」することである
- どういう式になると、より「単純」なのか
- 等価であることを保証する変換規則の集まりがあらかじめ定められている必要がある
  - → 計算体系

λ式の変換例

- 以下のλ式を考える
  λx M (A)
- Mがλy (xy)で、Aが(yy)というλ式のとき、上のλ式は
  λx (λy (xy)) (yy)
- これを評価すると以下のようになる…?
  λy ((yy)y)
- Aに含まれる変数yは自由変数であり、評価した結果、束縛変数になってしまった。おかしい。

変数の有効範囲

- 前頁の例がおかしいのは、Mに含まれる変数yは束縛変数であり、Aに含まれる変数yは自由変数であり、それぞれ別の変数を指しているにもかかわらず、評価の際にそれを混同してしまった点にある
- つまり、個々の変数には有効範囲があり、それを越えた場所に同じ名前の変数があっても、それは別の変数として明確に区別する必要がある

λ式の「計算」

- λ式を「計算する」とは、そのλ式を別のλ式へ「変換」することである
- λ-calculusの体系では3つの「変換」規則が定められている
  - α変換（α-conversion）
  - β変換（β-conversion）
  - η変換（η-conversion）

substitution: 式中の変数の置換

- λ式の変換は、式に現れる変数を他のものと置き換えることが基本
- この置き換えのことをsubstitutionと呼ぶ
- substitutionは、一般に「代入」と訳されるが、手続型言語における「代入」とは意味が異なる
  - 手続型言語における「代入」は、assignmentの訳である
- むしろ、「置換」に相当する

substitution rule: 置換規則

- N,Zをλ式、xを変数とする
- N中のxをZで置き換えるsubstitutionを
  [Z/x] N
  と表すことにする
- たとえばλz (xz)の変数xをλy (y)で置き換えるsubstitutionは
  [λy (y)/x] λz (xz)
  であり、これは
  λz ((λy (y)) z)
  になる

substitution ruleの厳密な定義

- λ式N中の変数xをλ式Zで置き換えた [Z/x] NをN'とする
- 以下のそれぞれの場合のsubstitutionの結果を定義する
  - Nがある変数だけの時
  - Nがλy Mの形になる時
  - NがF (A)の形になる時

substitution: Nがある変数だけの時

- Nがxの時、N'はZ

  ex. [Z/x] x ➔ Z

- Nがxでない時、N'はNのまま

  ex. [Z/x] y ➔ y

substitution: Nが λy Mの形の時 (1)

- yがxと同じ変数である時、N'はNのまま

  ex. [Z/x] λx x ➔ λx x

substitution: Nが λy Mの形の時 (2)

- yがxと異なる変数で
  - xがMの自由変数でないか、あるいは、yがZの自由変数でないならば、N'は λy [Z/x] M
    ex. [λy y/x] λy xy
    ➔ λy [λy y/x] xy ➔ λy ((λy y)y)
  - xがMの自由変数で、かつ、yがZの自由変数の時、N'は λz [Z/x] ([z/y] M)
    ex. [yy/x] λy xy
    ➔ λz [yy/x] ([z/y] xy)
    ➔ λz [yy/x] (xz) ➔ λz yyz
    - ただし、zはNにもZにも現れない変数

substitution: Nが F (A)の形の時

- N'は[Z/x] F ([Z/x] A)
  - ex. Fがλx xxで、Aがλy xyとすると、
    Nは(λx xx) (λy xy)となる。
    Zがzzの時、N'つまり[Z/x] Nは、
    [zz/x] (λx xx) (λy xy)
    ➔ (λx xx) ([zz/x] (λy xy))
    ➔ (λx xx) (λy zzy)

α変換 (α-conversion)

- **束縛変数の名前の置換**

$$\lambda x\ M \overset{\alpha}{\Longleftrightarrow} \lambda y\ [y/x]\ M$$

(**ただし、**y**が**M**の自由変数ではないとする**)

β変換 (β-conversion)

- **関数の適用対象による束縛変数の置換**

$$\lambda x\ M\ (N) \overset{\beta}{\Longleftrightarrow} [N/x]\ M$$

η変換（η-conversion）

- **外延**

$$\lambda x\ (M\ (x)) \overset{\eta}{\Longleftrightarrow} M$$

(**ただし、x**はM**の自由変数ではないとする**)

簡約 (reduction)

- β変換およびη変換において、左から右方向への変換を、それぞれ、β簡約 (β-reduction)、η簡約 (η-reduction) と呼ぶ
  - 縮約とも呼ぶ
- λ式Vがβおよびηを何回かおこなった結果、λ式Wとなるとき、以下のように表記する

$$V \xrightarrow{\beta} W$$

$$V \xrightarrow{\eta} W$$

被簡約子 (redex)

- β簡約の対象となる
  λx M (N)
  や、η簡約の対象となる
  λx (M (x))
  という形のλ式のことを、被簡約子と呼ぶ
- β簡約の対象となるλ式をβ被簡約子(β-redex)、η簡約の対象となるλ式をη被簡約子(η-redex)と呼ぶ
  - β基、η基とも呼ぶ

コントラクタム (contractum)

- redexを簡約して得られるλ式をコントラクタムと呼ぶ
- β-redexであるλx M (N)のコントラクタムは[N/x] Mとなる

既約形 (normal form)と評価

- redexを部分式に持たないλ式、つまり、これ以上の簡約ができないλ式を既約形と呼ぶ
  - 正規形とも呼ぶ
- β-redexを持たないλ式、つまり、これ以上のβ簡約ができないλ式をβ既約形と呼ぶ
- η-redexを持たないλ式、つまり、これ以上のη簡約ができないλ式をη既約形と呼ぶ
- β-redexもη-redexも持たないλ式、つまり、これ以上のβ簡約もη簡約もできないλ式をβη既約形と呼ぶ
- あるλ式の「評価」とは、したがって、λ式に0回以上の簡約をおこなって既約形を求めることである

既約形についてのいくつかの疑問

- いかなるλ式でも既約形は存在するのか?
- あるλ式に既約形が存在するとき、必ず既約形は一つしか存在しないのか?
  - 言い換えると、そのλ式のどの部分にどの変換/簡約をどのような順序でおこなうかによって、到達する既約形が異なることはないのか?
- あるλ式に既約形が一つだけ存在する場合、そのλ式のどの部分にどの変換/簡約をどのような順序でおこなっても、必ずその既約形に到達するのか?

既約形は必ずあるのか?

- ない場合もある
- 例) $(\lambda x\ x\ (x))\ (\lambda x\ x\ (x))$
- これにはβ既約形が存在しない

$$(\lambda x\ x\ (x))\ (\lambda x\ x\ (x))$$

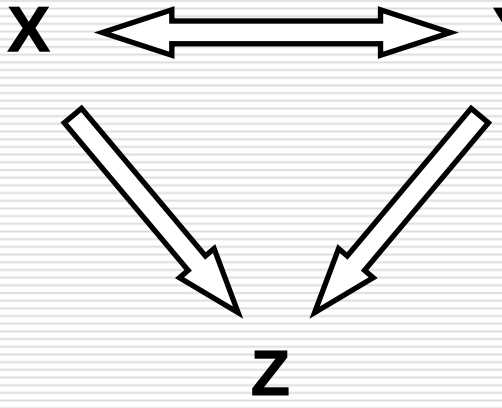
$$\xrightarrow{\beta} (\lambda x\ x\ (x))\ (\lambda x\ x\ (x))$$

$$\xrightarrow{\beta} (\lambda x\ x\ (x))\ (\lambda x\ x\ (x))\ \ \ldots$$

- β簡約した結果が同じλ式になってしまうので、β既約形が存在しない (しかも上の例は停止しない)
- ちなみに上の式はη簡約がこれ以上おこなえないのでη既約形になっている

既約形は(あるなら)一つしかないか?

- チャーチ・ロッサー性 (Church-Rosser property) もしくは合流性 (confluence) という
- 二つのλ式X,Yが、α変換/β変換/η変換で互いに変換可能である場合、α変換/β簡約/η簡約によって、XからもYからも簡約可能なλ式Zが存在する



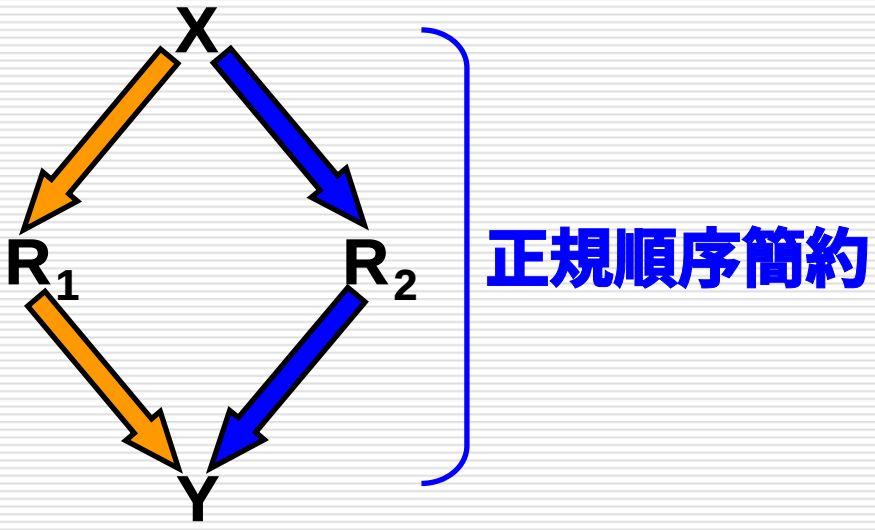
- この性質は、型なしλ算法の計算体系では成立するが、型付きλ算法の計算体系では成立しない

どういう順序で簡約しても既約形になるか?

- 以下のλ式を考える
  λx λy (y (y)) ((λx (x (x))) (λx (x (x))))
- このλ式のβ簡約は2通りある
- [1] λx λy (y (y)) の束縛変数xを ((λx (x (x))) (λx (x (x)))) で置き換える
  - 簡約結果はλy (y (y)) となり、これはβ既約形である
- [2] λx (x (x)) の束縛変数xを λx (x (x)) で置き換える
  - 簡約が停止しないため、β既約形にならない
- つまり、λ式によっては、既約形になる簡約順序と、ならない簡約順序がある

既約形になる簡約順序はどういうものか?

- ある λ 式Xを既約形Yに簡約可能なα変換/ β簡約/ η 簡約の適用順序が存在するなら、正規順序簡約戦略によってXからYに簡約することができる



正規順序簡約戦略

- **正規順序簡約戦略** (normal order reduction strategy)
  - **簡約戦略** (reduction strategy) **の一つ**
  - **最左簡約戦略** (leftmost reduction strategy) **とも言う**
  - **最も左側にある**redex**を**contractum**に置き換える簡約戦略**
- **プログラミング言語の引数受渡機構でいうと、**Call-by-Name**がそれに近いといえる**

正規順序簡約戦略によるλ式の評価

- 例)

(λx (λy y (x)) (x)) (λz z)
(λy y (λz z)) (λz z)
(λz z) (λz z)
λz z

作用的順序簡約戦略

- **作用的順序簡約戦略** (application order reduction strategy)
  - F (A)**というλ式がある時に、**Aの評価をまず先に**おこなってから、その結果に対して**Fを適用する、**という簡約戦略**
  - strict reduction strategy**と同じ**
- **プログラミング言語の引数受渡機構でいうと、**Call-by-Value**がそれに近いといえる**