Type Systems and Programming Languages
第5章: The Untyped Lambda-Calculus

斉藤 新 [shin]

May 10, 2001

Outline

- Introduction: λ-calculusとは?
  - 数学との違いは?
- Syntax
  - Abstract syntax
  - 言語の定義とtermの構造に関する帰納法
- Semantics
  - α/β-conversion
  - 評価戦略
- λ-calculusによるProgramming
  - 多引数関数と高階関数: Currying
  - 真偽値/自然数/データ構造 の表現
  - 不動点演算子と再帰関数

λ-calculus (λ計算)とは?

Alonzo Churchらによって提唱された計算モデル (1920s–1930s)

- 歴史: もともとは
  数理論理学におけるモデル: effective に計算可能な関数とは?
  - 帰納的関数
  - Turing機械によって計算可能な関数
  - λ-calculusで表現可能な関数

  これらの関数の集合は一致する →
  Church's Thesis:「これらを計算可能関数としよう」
- 「関数」が計算の本質: すべてのentityは関数である
  - 単純な syntax: 関数を作る/関数を適用する
  - 十分な表現力: 自然数、真偽値、pairやリストなど
- 関数型言語のCore Calculusとして採用

λ抽象(λ-abstraction)の概念

数学では例えば1を足す関数(successor関数)は

$$succ(x) \triangleq x + 1$$

のように定義して

$$succ(3) \Rightarrow 4$$

のように適用する。

だが関数名は計算の性質を考えるときには本質的ではない。
→「xを受け取って$x+1$を返す」という操作(関数)を直接定義する

$$succ \triangleq \lambda x.\,(x + 1)$$

このような関数の定義方法をλ-abstraction (λ抽象)という

こうすると、$succ$を定義しなくても直接$succ(3)$を計算できる

$$(\lambda x.\,(x + 1))(3) \Rightarrow 4$$

数学とλ計算の共通点/相違点

- 共通点:
  - 関数を関数に適用できる
    関数の値が関数であってもよい

$$\frac{d}{dx}(x^2) = 2x$$
$$deriv_x(\lambda x.\, x^2) = \lambda x.\, 2x$$

- 相違点:
  - 関数に自分自身を適用できる

$$id \triangleq \lambda x.\, x$$
$$id(id)$$

数学では禁止される。λ計算ではOKで、結果はidつまり$\lambda x.\, x$

実際、λ計算ではすべての存在は関数である。
→ 自然数などもすべて関数として表す(表せる)。

λ-calculusのsyntax

λ式を表す文字列の集合(言語という)をBNF (Backus-Naur Form)で定義する。

$t :=$	項、λ式:
x	変数
$\lambda x.\, t$	λ抽象
$t\, t$	関数適用

注意:

- 上のxやtはメタ変数である。他に変数/項を表すメタ変数として

$$x, y, z, \ldots$$
$$s, t, u, \ldots$$

  を用いる。
- 上で与えたsyntaxはabstract syntaxと呼ばれる(後述)。

Abstract syntax

前頁で与えた文法は(普通に解釈すると)曖昧である。なぜなら$\lambda x.\, x\, y$は

$$\lambda x.\, (x\, y)$$
$$(\lambda x.\, x)\, y$$

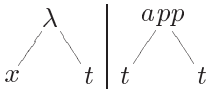
の両方に解釈できるからである。

なので、厳密にはsyntaxに括弧 "(" と ")" を導入して

$$t := x \mid (\lambda x.\, t) \mid (t\, t)$$

としなければならないが、これは煩雑である。

前頁のように書いたときには、λ式の文字列を定義しているというよりは、それをparseした後の構文木を定義していると考える。
これを(concrete syntaxと対比して) abstract syntaxという。

$t :=$ x |  |

Abstract syntax (contd.)

括弧は構文木を指定するためにわれわれが適宜補うことにするが、ある程度決まりを設けておく。

- 関数適用は左結合とする。つまり、$s\, t\, u$ は $(s\, t)\, u$ と解釈
- λ抽象の本体部はできるだけ右に長く解釈する。
  つまり、$\lambda x.\, \lambda y.\, x\, y$ は $\lambda x.\, (\lambda y.\, (x\, y))$ と解釈

なので、λ計算では $succ(3)$ とは書かないで、$(succ\ 3)$ と書く。

ちなみに、文献によっては

- $\lambda x.\, \lambda y.\, t$ を $\lambda xy.\, t$ と書く

としていることもある。

λ式をformalに定義する

これから、λ式の集合 $\mathcal{T}$ をformalに定義する。

$\mathcal{V}$ を変数名の(可算)集合であるとする。
$\mathcal{T}$ は以下の条件を満たす最小の集合である。

1. $\forall x\ (x \in \mathcal{V} \Rightarrow x \in \mathcal{T})$
2. $\forall x, t_1\ (x \in \mathcal{V} \wedge t_1 \in \mathcal{T} \Rightarrow \lambda x.\, t_1 \in \mathcal{T})$
3. $\forall t_1, t_2\ (t_1 \in \mathcal{T} \wedge t_2 \in \mathcal{T} \Rightarrow (t_1\ t_2) \in \mathcal{T})$

このとき $\mathcal{T}$ は以下の集合 $\mathcal{S} = \bigcup_{i \in \mathbb{N}} \mathcal{S}_i$ と等しいことがポイントである。

(集合列 $\{\mathcal{S}_i\}_{i \in \mathbb{N}}$ は単調増加)

$$\begin{aligned} \mathcal{S}_0 &= \emptyset \\ \mathcal{S}_{i+1} &= \mathcal{V} \\ &\quad \cup \{\lambda x.\, t_1 \mid x \in \mathcal{V},\ t_1 \in \mathcal{S}_i\} \\ &\quad \cup \{t_1\ t_2 \mid t_1 \in \mathcal{S}_i,\ t_2 \in \mathcal{S}_i\} \end{aligned}$$

構造に関する帰納法

$\mathcal{T} = \mathcal{S} = \bigcup_{i \in \mathbb{N}} \mathcal{S}_i$ であるから、
すべての項 $t \in \mathcal{T}$ に関してある命題 $P(t)$ が成り立つことをいうには

1. $P(t)$ が $t \in \mathcal{S}_0$ なるすべての項 t に対して成立
2. $P(s)$ が $s \in \mathcal{S}_i$ なるすべての項 s に対して成立 $\Rightarrow$
   $P(t)$ が $t \in \mathcal{S}_{i+1}$ なるすべての項 t に対して成立

を示せば、数学的帰納法によって $P(\mathcal{S}_i)$ がすべての i に対していえる。
よって $P(\mathcal{T})$ が厳密に(数学的に/論理学的に)いえる。

これを構造に関する帰納法という。

関数の構造に関する帰納的定義

また、$\mathcal{T}$ から(たとえば)自然数の集合 $\mathbb{N}$ への関数 f を定義するには

1. $t \in \mathcal{S}_0$ に対して $f(t)$ を定義する
2. $s \in \mathcal{S}_i$ に対して $f(s)$ が定義されているとして、
   これを用いて $t \in \mathcal{S}_{i+1}$ に対して $f(t)$ を定義する

こう定義するとまた $\mathcal{T} = \mathcal{S} = \bigcup_{i \in \mathbb{N}} \mathcal{S}_i$ であることにより
f が $\mathcal{T}$ から $\mathbb{N}$ への関数(つまり、$\mathcal{T} \times \mathbb{N}$ の部分集合)として
厳密に定義される。

これが関数の、構造に関する帰納的定義である。

束縛/自由変数

$\lambda x.\, (x + y)$ において x は関数の仮引数になっている変数である。
このような変数をbound variable (束縛変数)という。

束縛変数でない変数をfree variable (自由変数)という。
上の例で y は自由変数である。

項 $(\lambda x.\, x)\ x$ では x は束縛変数としても自由変数としても出てきている。
この場合は occurence (出現)という言葉を用いて区別する。
一つめの x (青)は束縛された出現であり、二つめ(赤)は自由な出現である。

Formal には項 t に対して自由変数の集合 $FV(t)$ を以下のように定義する。

$$\begin{aligned} FV(x) &\triangleq \{x\} \\ FV(\lambda x.\, t) &\triangleq FV(t) \setminus \{x\} \\ FV(t_1\ t_2) &\triangleq FV(t_1) \cup FV(t_2) \end{aligned}$$

自由変数を持たない項はclosed (閉じている)/combinatorとよばれる。

Operational Semantics (操作的意味)

Pureなλ計算は数字などの値を全く持たないため、
計算に関しては基本的には「関数を適用する」をどうやるかだけである。

$$(\lambda x.\, t_1)\ t_2 \to [x \mapsto t_2]\ t_1$$

$[x \mapsto t_2]\ t_1$ は t_1 の中の自由変数 x を項 t_2 で置き換えることを意味する。

$$(\lambda x.\, x)\ y \to y$$
$$(\lambda x.\, x\ y)(u\ r) \to (u\ r)\ y$$

一般にこのような評価をreduction (簡約)とよぶ。
上の簡約は β-reduction と呼ばれる。

また、$(\lambda x.\, t_1)\ t_2$ の形の λ式(または部分式)をredex (reduction expression)という。

実は上の置き換えの説明は正確でない (理由は次頁)。

変数の置き換え (1)

前頁の説明が不十分なわけ:

$\lambda y.\, x$ は引数として何を受け取ってもある x を返す関数である。
これに対して x を y で置き換える。
結果として欲しいのは、引数として何を受け取っても y を返す関数である。

$$[x \mapsto y]\ (\lambda y.\, x) = \lambda y.\, y$$

ところがこれは恒等関数であり、始めの関数とは異なる。

原因は青字の y は同一のものであるのに、始め自由であった y が束縛されてしまっていることにある。これをvariable captureという。

よって、$[x \mapsto t_2]\ t_1$ は capture-avoiding substitution (束縛を回避するような置き換え)として定義する必要がある。

そのためには、束縛変数は違う変数に置き換えてもよいことを利用する。

変数の置き換え (2)

$\lambda x.\, x$ と $\lambda y.\, y$ はどちらも恒等関数である。

一般に、束縛変数を違う変数に置き換えても項の意味は変わらないと考えてよい。

束縛変数名を付け替える変換を α-conversion (α変換)といい、α変換で互いに移りあう項をα-equivalentであるという。

$$\lambda x.\, x =_\alpha \lambda y.\, y$$

ここではα変換で移りあう項は同一視し、α変換を任意の場所で行ってよいとする。前頁の例では束縛変数yをここに出てこない変数zに置き換えて

$$\begin{aligned}[x \mapsto y]\ (\lambda y.\, x) &=_\alpha [x \mapsto y]\ (\lambda z.\, x) \\ &= \lambda z.\, y\end{aligned}$$

これらの取り決めのもとで、変数の置き換えを定義する (次頁)。

変数の置き換え (3)

「変数xの項sによる置き換え」を以下のように定義する (x, y などはメタ変数であることに注意)。

$$\begin{aligned}
[x \mapsto s]\ x &\triangleq s \\
[x \mapsto s]\ y &\triangleq y && x \neq y \\
[x \mapsto s]\ (\lambda y.\, t) &\triangleq \lambda y.\, [x \mapsto s]\ t && x \neq y \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ t_2) &\triangleq ([x \mapsto s]\ t_1)\ ([x \mapsto s]\ t_2)
\end{aligned}$$

3行目が表すのは、$[x \mapsto s]\ (\lambda x.\, t)$ のような場合は束縛変数xを、

$$y \neq x \wedge y \notin FV(s)$$

を満たすような変数yに置き換えてから置換を行うということ。

計算機上で実装する場合、この操作は非常に重い。
ので、これを回避するようなλ項の表現方法 (nameless表現) が考えられている (次章)。

評価戦略

一般に、項の中には複数のredexがある。そのうちどれを簡約するかを定める規則をreduction strategy(評価戦略)という。
どの戦略を選ぶかによって項について成り立つ性質が変わるので注意が必要である。

主な評価戦略:

- full β-reduction
- normal order (leftmost) 戦略
- call-by-name (lazy) 戦略
- call-by-need 戦略
- call-by-value (eager) 戦略

これらについて、項 $id\ (id\ (\lambda z.\ id\ z))$ がどのように簡約されるか見ていくことにする。

Full β-reduction 戦略

任意のredexを簡約してよい。

$id\ (id\ (\lambda z.\ id\ z))$ では

$$id\ (id\ (\lambda z.\ id\ z))$$
$$id\ (id\ (\lambda z.\ id\ z))$$
$$id\ (id\ (\lambda z.\ id\ z))$$

の3通りの簡約の候補がある。例えば内側から簡約することにすると

$$\begin{aligned} id\ (id\ (\lambda z.\ id\ z)) &\to id\ (id\ (\lambda z.\ z)) \\ &\to id\ (\lambda z.\ z) \\ &\to \lambda z.\ z \end{aligned}$$

となる。もちろん他の簡約方法も存在する。

Normal order (leftmost) 戦略

(※ ここ以降の評価戦略は決定的である。計算機上で実装するときに都合がいいからである。)

この評価戦略はもっとも左(もっとも外側)にあるredexを簡約する。

$id\ (id\ (\lambda z.\ id\ z))$ では

$$\begin{aligned} id\ (id\ (\lambda z.\ id\ z)) &\to id\ (\lambda z.\ id\ z) \\ &\to \lambda z.\ id\ z \\ &\to \lambda z.\ z \end{aligned}$$

となる。最左戦略には

- 項 t が normal form (正規形) n を持つならば
  それは最左戦略で求まる

という性質がある。

Call-by-name (lazy) 戦略

もっとも左(もっとも外側)にあるredexを簡約するが、λ抽象の本体は簡約しない(普通関数本体は関数呼び出し時に評価したいから)。

$id\ (id\ (\lambda z.\ id\ z))$ では

$$\begin{aligned} id\ (id\ (\lambda z.\ id\ z)) &\to id\ (\lambda z.\ id\ z) \\ &\to \lambda z.\ id\ z \end{aligned}$$

で簡約はストップする。この戦略を用いている言語としては Algol [Naur$^+$ 63] などがある。この戦略は

- 関数適用 $(\lambda x.\ t_1)\ t_2$ において、引数 t_2 は評価しないで先に適用(置き換え)を行う。
- 関数本体 t_1 の評価で x が出てくるたびに t_2 の評価が行われる。

不必要な評価はしないので無限リストや特殊構文などを表現しやすいという利点があるが、仮引数が本体に多数出現すると効率が悪い。

Call-by-need 戦略

大体 call-by-name と同じだが、

$$(\lambda x.\, t_1)\ t_2$$

において、関数本体で x が最初に出てきたときは t_2 を評価するが、その結果を取っておいて x の2回目以降の出現では結果をそのまま使う。

call-by-name の効率の悪さを解決する評価戦略である。
これを用いている言語としては Haskell [Hudak$^+$ 92] などがある。

Call-by-value (eager) 戦略

左(外)から評価していくが、関数の本体は評価しない。
$(t_1\ t_2)$ において、引数 t_2 が値(つまりλ抽象)になるまで評価してから関数適用を行う。

$id\ (id\ (\lambda z.\, id\ z))$ では

$$\begin{aligned} id\ (id\ (\lambda z.\, id\ z)) &\to id\ (\lambda z.\, id\ z) \\ &\to \lambda z.\, id\ z \end{aligned}$$

この戦略を用いている言語は多数あり、ML [Gordon$^+$ 79] や Scheme [Sussman$^+$ 75] などがある。

本書では以下、この call-by-value を用いることにする。理由は

- 主流の言語で用いられている戦略である
- 例外や参照などの機構を導入するのが容易

などである。

操作的意味の定義

文法:

t	:=	項:
	x	変数
	$\lambda x.\, t$	抽象
	$t\ t$	適用
v	:=	値:
	$\lambda x.\, t$	λ抽象

評価規則:

$$t_1 \to t_1' \Rightarrow t_1\ t_2 \to t_1'\ t_2 \qquad \text{(E-App1)}$$
$$t_2 \to t_2' \Rightarrow v_1\ t_2 \to v_1\ t_2' \qquad \text{(E-App2)}$$
$$(\lambda x.\, t_1)\ v_2 \to [x \mapsto v_2]\ t_1 \qquad \text{(E-AppAbs)}$$

多引数関数の表現: Currying (1)

これからλ計算で自然数やデータ構造が表現できることを見ていくがその前に

λ計算には(普通)多引数関数はない!

ではどう表現するか?

例えば引数を2つ受け取ってその和を返す関数は

$$\lambda(x, y).\, x + y$$

と書くのではなくて

$$\lambda x.\, \lambda y.\, x + y$$

と書く。例えば $2 + 5$ を計算するときは

$$(\lambda x.\, \lambda y.\, x + y)\ 2\ 5$$

を計算すればよい。

多引数関数の表現: Currying (2)

$2+5$ を計算するときは $(\lambda x.\, \lambda y.\, x+y)\ 2\ 5$ を計算すればよい。
その心は

- 足し算する関数は第1引数 2 を受け取って関数 $\lambda y.\, 2+y$ を返す
- その関数は第2引数 5 を受け取って $2+5$ を返す。

関数を引数に取ったり返したりする関数を higher order function (高階関数) というが、多引数関数は高階関数を使って表現できる。

この、多引数関数 → 高階関数 という変換を currying (カリー化) という。
この逆の変換は uncurrying という。

注: currying を考案したのは Curry ではなく、Schönfinkel という人らしい。

真偽値の表現

データ構造のencodingを考えるときは、それに対する操作とともに考えるとわかりやすい。

真偽値は何に使うかというと、当然条件分岐 if b then t else f である。
これをλ式としては

$$test\ b\ t\ f \to^* t \text{ (if } b \text{ is true)},\ f \text{ (if } b \text{ is false)}$$

と評価されるような項 $test$ で表現したい。

すると、真(偽 resp.)は t と f のうち t (f resp.) を選ぶようなものであればよい。

よって、以下のencodingを得る。

$$tru \triangleq \lambda t.\, \lambda f.\, t$$
$$fls \triangleq \lambda t.\, \lambda f.\, f$$
$$test \triangleq \lambda b.\, \lambda t.\, \lambda f.\, b\ t\ f$$

真偽値に関する演算

次に真偽値に関する演算を考える。

b と c の論理積 (and) を求める。if 文で書くと b and c は

$$\text{if } b \text{ then } c \text{ else false}$$

だから、$test\ b\ c\ fls$ に等しい。つまり、

$$and \triangleq \lambda b.\, \lambda c.\, (b\ c\ fls)$$

である。同様に論理和演算や否定演算も定義できる。

$$or \triangleq \lambda b.\, \lambda c.\, (b\ tru\ c)$$
$$not \triangleq \lambda b.\, (b\ fls\ tru)$$

Pairの表現

真偽値を用いれば pair も簡単に表現できる。

$pair\ f\ s$ で pair を構成するとする。
pair に適用される演算は fst (第1成分)、snd (第2成分) である。
これを実現するため、pair は

- true を渡されたら第1成分を返す
- false を渡されたら第2成分を返す

とすればよいから

$$pair \triangleq \lambda f.\, \lambda s.\, \lambda b.\, (test\ b\ f\ s) = \lambda f.\, \lambda s.\, \lambda b.\, (b\ f\ s)$$
$$fst \triangleq \lambda p.\, (p\ tru)$$
$$snd \triangleq \lambda p.\, (p\ fls)$$

$pair\ f\ s = \lambda b.\, (test\ b\ f\ s)$ に注意する。

自然数の表現

Church numeral という自然数の encoding を紹介する。

これは 0 と succ を用いた自然数の表現と似ていて、数学的帰納法の概念を用いている encoding である。

$$c_0 \triangleq \lambda s.\, \lambda z.\, z$$
$$c_1 \triangleq \lambda s.\, \lambda z.\, s\ z$$
$$c_2 \triangleq \lambda s.\, \lambda z.\, s\ (s\ z)$$
$$\cdots$$
$$c_i \triangleq \lambda s.\, \lambda z.\, s^i\ z$$
$$\cdots$$

つまり自然数 n は s, z を受け取って z に s を n 回適用するような関数である。
(3章の言語では n は 0 に succ を n 回適用した項で表現されたことを思い出してほしい)

自然数に関する演算 (1)

前頁の考察をもとに、successor function (後継者関数) は以下のように定義できる。

$$scc \triangleq \lambda n.\, \lambda s.\, \lambda z.\, s\ (n\ s\ z)$$
$$scc2 \triangleq \lambda n.\, \lambda s.\, \lambda z.\, n\ s\ (s\ z)$$

どちらも $s^n\ z$ から $s^{n+1}\ z$ を構成している。

加算関数は以下のように定義できる。

$$plus \triangleq \lambda m.\, \lambda n.\, \lambda s.\, \lambda z.\, m\ s\ (n\ s\ z)$$
$$plus\ m\ n = \lambda s.\, \lambda z.\, m\ s\ (n\ s\ z)$$

つまり、$n\ s\ z\ (= s^n\ z)$ に s を m 回適用すると $s^{m+n}\ z$ になる。

scc を用いて、n に scc を m 回適用すると考えてもよい。

$$plus2 \triangleq \lambda m.\, \lambda n.\, m\ scc\ n$$

自然数に関する演算 (2)

乗算 $m \times n$ に関しては、c_0 に $plus\ n$ (これは1引数関数) を m 回適用すればよいから、

$$times \triangleq \lambda m.\, \lambda n.\, m\ (plus\ n)\ c_0$$

$plus$ を使わないとすれば、

$$times2 \triangleq \lambda m.\, \lambda n.\, \lambda s.\, \lambda z.\, m\ (n\ s)\ z$$

z に $(n\ s)$ を m 回適用すると考える。
($(n\ s)$ とは引数に s を n 回適用する関数である)

この関数はさらに

$$times3 \triangleq \lambda m.\, \lambda n.\, \lambda s.\, m\ (n\ s)$$

と簡単化できる (理由はちょっと複雑なので省略)。

自然数に関する演算 (3)

ゼロ検査:

- c_0 は値として z を返す
- c_i $(i \geq 1)$ は z に s を1回以上適用した結果を返す

から、z は tru にして s は fls を返す定数関数にすればよい。

$$iszro \triangleq \lambda n.\, n\ (\lambda x.\, fls)\ tru$$

Predecessor function prd: これは結構面倒。
z にある操作 s を n 回適用したときに、c_{n-1} を作る必要がある。

そのために s を、$\langle c_j, c_i \rangle$ から $\langle c_i, c_{i+1} \rangle$ を作る操作とする。
$z = \langle c_0, c_0 \rangle$ として s を適用していくと

$$z = \langle c_0, c_0 \rangle \xrightarrow{s} \langle c_0, c_1 \rangle \xrightarrow{s} \langle c_1, c_2 \rangle \xrightarrow{s} \cdots \xrightarrow{s} \langle c_{n-1}, c_n \rangle = s^n\ z$$

最後に pair の第1成分を返せばよい。
$m - n$ は m に prd を n 回適用すればよい。

演習問題の略解

- $prd\ c_n$ は前頁で見たように、$O(n)$ ステップの計算を行う。
- 等値判定: $m = n$ は $m \geq n \wedge n \geq m$ である。
  さらに、$m \geq n \Longleftrightarrow n - m = 0$ であることを使う。
- リストの定義: リストの構造による場合分けは以下のように行う。

  $$l\ c\ n$$

  l はリスト、c はリストが cons セルのときに計算を行う関数、
  n はリストが nil (空リスト) のときの値

  これより表現は以下のようになる。

  $$nil \triangleq \lambda c.\, \lambda n.\, n$$
  $$cons \triangleq \lambda h.\, \lambda t.\, c\ h\ (t\ c\ n)$$
  $$(cons\ h\ t \rightarrow^* c\ h\ (t\ c\ n))$$

  cons セルを関数 c で計算するときは、tail 部分をまず計算して $(t\ c\ n)$ その結果 と head h に c を適用する。

再帰関数 (1)

λ 計算では (第3章の言語と異なり) すべての項が正規形に到達するとは限らない。正規形を持たない項は diverge (発散) するという。

例として以下のような combinator がある。

$$\Omega \triangleq (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

redex は1つしかなく、簡約を行うと元の項と同じになる。よって発散する。

これがなんの役に立つかというと、再帰関数を表現するための fixed-point (不動点) combinator を構成するためのヒントになる。

用語: 関数 f の不動点とは $f\ x = x$ を満たす x のことである。
もちろん、必ず存在するとは限らない。

Primitiveなデータ型

見てきたように、
pure な λ 計算でも自然数や真偽値やその他のデータ型を定義できる。
しかし実際の言語に組み込むときや研究の上ではこれらを primitive な値として λ 計算に組み込むことが多い。

λ 計算に第3章の自然数、真偽値を組み込んだものを λNB で表す。
primitive な値と Church encoding による表現は互いに変換可能である。

こうする理由の一つ: 評価戦略
call-by-value のもとでは、$times\ c_2\ c_2 \rightarrow^* c_4$ とはならない。
(ただし full β-reduction のもとでは成り立つ。
また、call-by-value のもとでも behaviorally equivalent (振る舞い的に等価) ではある。この等価の定義はここでは述べない)

前述の定義が正しいことは、たとえば $plus\ c_i\ c_j$ と c_{i+j} が振る舞い等価であることなどを示せば証明できる。

再帰関数 (2)

Pure な λ 計算に関しては必ず不動点が存在する。

Call-by-value のための不動点 combinator (Y-combinator) は以下のように定義される。

$$fix \triangleq \lambda f.\, (\lambda x.\, f\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, f\ (\lambda y.\, x\ x\ y))$$

これにより $fix\ f = f\ (fix\ f)$ を満たすことが確かめられる。
($fix\ f \rightarrow^* f\ (fix\ f)$ とはならない。$fix\ f \rightarrow^* f\ (\lambda y.\, (fix\ f)\ y)$ となる)

簡単な (というか無意味な) 再帰関数の例をあげて計算してみる。

$$f = \lambda n.\, f\ (n - 1)$$

で表される関数 f を計算する。この式自体は右辺にも f が出てくるので f の定義というよりは f が満たすべき等式である。

ここで、$F \triangleq \lambda f.\, \lambda n.\, f\ (n - 1)$ と定義して $f' = fix\ F$ とおくとこの f' が f が満たすべき等式を満たすことがわかる。

再帰関数 (3)

$F \triangleq \lambda f.\, \lambda n.\, f\ (n-1)$ で $f' = fix\ F$ だから

$$
\begin{aligned}
(f'\ n) &= (fix\ F)\ n \\
&\to (\lambda x.\, F\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, F\ (\lambda y.\, x\ x\ y))\ n \\
&\to F\ (\lambda y.\, (\lambda x.\, F\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, F\ (\lambda y.\, x\ x\ y))\ y)\ n \\
&= F\ (\lambda y.\, (fix\ F)\ y)\ n \\
&\to (\lambda n.\, (\lambda y.\, (fix\ F)\ y)\ (n-1))\ n \\
&\to (\lambda y.\, (fix\ F)\ y)\ (n-1) \\
&\to (fix\ F)\ (n-1) \\
&\to^* F\ (\lambda y.\, (fix\ F)\ y)\ (n-1) \to \ldots
\end{aligned}
$$

として再帰関数を計算することができる。テキストには階乗を計算する例が載っているので参考にされたい。

不動点combinatorは他の評価戦略を用いるときはもっと簡単。
Call-by-value では、$(fix\ F)$ の評価を遅らせるために複雑になっている。

おまけ

- λ式について: 昔は、$\lambda x.\, t$ において t の中に x が少なくとも1回現れないと正しい項とみなされなかった。
- 「α変換」などの言葉について: Church の文献に項に対する変換として
  1. I-変換: 束縛変数のrenaming
  2. II-変換: $(\lambda x.\, s)\ t$ を $[x \mapsto t]\ s$ に変換する
  3. III-変換: $[x \mapsto t]\ s$ を $(\lambda x.\, s)\ t$ に変換する

  が挙げられていたことが由来と思われる。なお、η変換という変換があるが、名前の由来はわからない。