ラムダ計算入門

2005 年度「計算機ソフトウェア工学」授業資料

住井 英二郎

sumii@ecei.tohoku.ac.jp

2005 年 6 月 24 日

ML をはじめ、ほとんどの（高階関数のある）関数型言語は、λ**計算**という計算モデルにもとづいて設計・開発されている。「λ 計算を知らないと関数型言語を使えない」というわけではないが、計算機科学の理論としては常識なので、「大学で情報科学／情報工学を専攻した」と言っても恥ずかしくないように、必要最低限のことをやっておこう。

1 型なし λ 計算

1.1 λ 式の構文

λ 計算には様々な拡張があるが、すべての基礎となる、もっとも**純粋**な λ 計算の抽象構文は以下のように定義される。

$$\begin{array}{rcll} e\ (\lambda\text{式}) & ::= & x & (\text{変数}) \\ & | & \lambda x.e & (\lambda\text{抽象}) \\ & | & e_1 e_2 & (\text{関数適用}) \end{array}$$

変数と関数適用の意味は ML と同様である。$\lambda x.e$ は、Objective Caml の `fun x -> e` と同じで、「x を受け取り、e の値を返す」関数である。

関数型言語の計算モデルとかいって、関数しかないじゃん！
（整数とか `let` はどこへいったの？）

と思うかもしれないが、実は整数や `let` なども、上のような λ 式だけで表せる[1]。たとえば、

```
let x = e1 in e2
```

は

$$(\lambda x.e_2)e_1$$

と表せる。どちらも、まず e_1 が x に代入され、それから e_2 が実行されるためである。このように、λ 計算は定義が極端に簡単（構文はたったの 3 行）であるにもかかわらず、普通のプログラミング言語と同等の計算能力があるので、基礎研究などでは非常に便利である。

なお、括弧が多すぎると読みづらいので、$\lambda x.(\lambda y.e)$ という形の式は $\lambda x.\lambda y.e$ と書く。また、$\lambda x.(e_1e_2)$ という形の式を $\lambda x.e_1e_2$ と略す。つまり、関数適用は λ 抽象より優先順位が高い。さらに、$(e_1e_2)e_3$ という形の式は $e_1e_2e_3$ と書く。つまり、関数適用は**左結合的**である。これは $10-3-7$ と書いたら $10-(3-7)$ ではなく $(10-3)-7$ になるのと同じことである。

1.2 β 簡約

構文だけ定義しても、それが何を意味するか定めないと、言語として成立しない。λ 式の意味は様々な方法で定義できるが、ここではもっとも一般的な small-step 操作的意味論の一つを用いることにする。

純粋な λ 計算には関数（λ 抽象）しか値がなく、後で見るように組や数などもすべて λ 抽象で表される。したがって、すべての計算は関数適用として表現される。

一般に、関数 $\lambda x.e_1$ を引数 e_2 に適用したら、まず x に e_2 が代入され、それから e_1 が実行されるはずである。これを

$$\frac{}{(\lambda x.e_1)e_2 \quad \to \quad [e_2/x]e_1} \qquad \text{(R-Beta)}$$

という規則により定義される、λ 式と λ 式の間の二項関係 $\to$ で表すことにしよう。ただし $[e_2/x]e_1$ は、e_1 の変数 x に e_2 を代入した式とする。この二項関係のことを λ 式の（1 ステップの）β **簡約**といい、

$$(\lambda x.e_1)e_2 \text{ は } [e_2/x]e_1 \text{ に（1 ステップで）} \beta \text{ 簡約される}$$

[1] それどころか、**およそコンピュータで可能なすべての計算は、λ 計算でも表現できる**。厳密にいうと、λ 計算は「チューリング完全」である。つまり、チューリング・マシンで可能な計算は、λ 計算でも表現できるし、逆も言える。

などという。

たとえば

$$(\lambda x.x)y \quad \to \quad y$$

であり、

$$(\lambda x.\lambda y.x)z \quad \to \quad \lambda y.z$$

である。また、

$$(\lambda x.x)(\lambda y.y) \quad \to \quad \lambda y.y$$

となる。

問 1. 上の他に（1 ステップの）β 簡約の例を挙げよ。 □

1.3 部分式の β 簡約

では、$(\lambda a.b)cd$ という λ 式は簡約できるだろうか。つまり、

$$(\lambda a.b)cd \quad \to \quad e$$

となるような式 e はあるだろうか[2]。先の規則 (R-Beta) は、適用される関数が λ 抽象の形になっていることを要求している。ところが、今は適用される関数がふたたび $(\lambda a.b)c$ という関数適用の形をしているので、先の規則 (R-Beta) を当てはめることができない。つまり、(R-Beta) だけでは、どうやっても $(\lambda a.b)cd$ を簡約することができない。これは、直感に反しており、おかしい。たとえば、Objective Caml における `(fun a -> b) c d` の意味とも合わない。

この問題を解決するためには、一般に関数適用 e_1e_2 で、適用される関数 e_1 を簡約する規則 (R-App1) が必要となる。

$$\frac{e_1 \quad \to \quad e_1'}{e_1e_2 \quad \to \quad e_1'e_2} \qquad \text{(R-App1)}$$

この規則があれば、たとえば先の例の λ 式も

$$(\lambda a.b)cd \quad \to \quad bd$$

[2] ここで $(\lambda a.b)cd$ は、$(\lambda a.b)(cd)$ ではなく $((\lambda a.b)c)d$ の略記だったことに注意せよ。したがって、$(\lambda a.b)cd = (\lambda a.b)(cd) \to b$ という簡約はありえない。

と簡約できる。なぜならば、

$$\dfrac{\dfrac{}{(\lambda a.b)c \quad \to \quad b}\text{R-Beta}}{(\lambda a.b)cd \quad \to \quad bd}\text{R-App1}$$

という導出がちゃんと存在するからである。

なお、(R-App1) と同様の理由により、関数適用 e_1e_2 において引数 e_2 を簡約する規則 (R-App2) も必要である。

$$\dfrac{e_2 \quad \to \quad e_2'}{e_1e_2 \quad \to \quad e_1e_2'} \qquad \text{(R-App2)}$$

問 2. (R-App2) を用いた簡約の例を挙げよ。また、その簡約の導出を書き下せ。 □

注 1. (R-App1) や (R-App2) は、関数適用 e_1e_2 において e_1 や e_2 を簡約するために必要であった。では、λ 抽象 $\lambda x.e$ において e を簡約する、以下のような規則 (R-Abs) は必要だろうか？

$$\dfrac{e \quad \to \quad e'}{\lambda x.e \quad \to \quad \lambda x.e'} \qquad \text{(R-Abs)}$$

これは実は「場合による」というのが答えである。たとえば、ML のプログラムを普通に実行する場合は、`fun x -> e` の `e` は関数が適用されるまで評価されないので、(R-Abs) のような簡約は不要である。しかし、コンパイラによる最適化や、関数と関数の等価性を議論する場合は、(R-Abs) のような簡約も必要となる。たとえば、$\lambda x.(\lambda y.y)x$ は $\lambda x.x$ に簡約できるから、前者を後者におきかえてもよい、などである。この入門では (R-Abs) も規則に含めて考えることにする。 □

1.4 自由変数、束縛変数、α 変換（重要ポイント！）

次に以下のような簡約を考えてみる。

$$(\lambda x.\lambda y.xy)y \quad \to \quad ???$$

規則 (R-Beta) を素朴に使用すると

$$(\lambda x.\lambda y.xy)y \quad \to \quad \lambda y.yy$$

となってしまうが、これはおかしい。なぜなら、

式 $(\lambda x.\lambda y.xy)y$ において、一つ目・二つ目の y と、三つ目の y は別々の変数である

はずなのに、上のように簡約した式 $\lambda y.yy$ においては、同一の変数になってしまっているからである。つまり、そもそも上のような簡約は誤っている。

では、どうすれば良いのか？　先の式

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

において、一つ目と二つ目の y はただの仮引数であるから、別に必ず y という名前である必要はない。そこで、一つ目と二つ目の y を新しい変数 y' におきかえた式

$$(\lambda x.\lambda y'.xy')y$$

を考える。この式を簡約すると

$$(\lambda x.\lambda y'.xy')y \quad \rightarrow \quad \lambda y'.yy'$$

となる。

上の y' のように、λ 抽象の仮引数となっている変数のことを**束縛変数**といい、

式 $(\lambda x.\lambda y'.xy')y$ において、変数 y' は $\lambda y'$ によって束縛されている

などという。逆に、束縛されていない変数のことを**自由変数**という。たとえば、

式 $(\lambda x.\lambda y'.xy')y$ において、変数 y は自由である

などという。

一般に、束縛変数の名前をつけかえても、λ 式の意味はかわらない（ただし、同一の束縛変数の名前は同時につけかえる）。たとえば、$\lambda x.xx$ と $\lambda y.yy$ は同一の関数である。このように束縛変数の名前をつけかえることを α **変換**という。λ 式 e_1 が e_2 に α 変換できることを「e_1 は e_2 と α **同値**である」という。

一方で、自由変数の名前をつけかえたら、λ 式の意味はかわってしまう。たとえば、$\lambda x.y$ と $\lambda x.z$ は

```
# let y = (fun a -> fun b -> a) ;; (* y を適当に定義しておく *)
val y : 'a -> 'b -> 'a = <fun>
# let z = (fun a -> fun b -> b) ;; (* z を適当に定義しておく *)
val z : 'a -> 'b -> 'b = <fun>
```

```
# let e1 = (fun x -> y) ;; (* λ x. y *)
val e1 : 'a -> 'b -> 'c -> 'b = <fun>
# let e2 = (fun x -> z) ;; (* λ x. z *)
val e2 : 'a -> 'b -> 'c -> 'c = <fun>
# e1 "xxx" "yyy" "zzz" ;;
- : string = "yyy"
# e2 "xxx" "yyy" "zzz" ;;
- : string = "zzz"
```

のように振る舞いが異なる。

注 2. 自由変数・束縛変数の概念は、λ 計算だけでなく数理論理学一般に共通である。たとえば、数学でも

$$f(x) \quad = \quad x+1$$

という関数 f と

$$g(y) \quad = \quad y+1$$

という関数 g は（仮引数の変数名 x, y が異なっていても）等しいとみなされるし、

$$\int \sqrt{1-x^2}dx$$

と

$$\int \sqrt{1-y^2}dy$$

は同じ式であるとみなされる。これは、x や y が束縛変数だから、α 変換しても数式の意味が変化しないためである。 □

β 簡約の場合だけでなく、一般に代入 $[e_2/x]e_1$ を行う際は、e_1 の束縛変数が e_2 の自由変数と衝突しないように、e_1 を α 変換する必要がある。このような代入は、e_1 の抽象構文についての場合わけと再帰により、以下のように定義できる。

$$\begin{array}{lcl}
[e_2/x]y & = & \begin{cases} e_2 & (x = y \text{ の場合}) \\ y & (x \neq y \text{ の場合}) \end{cases} \\
[e_2/x](\lambda y.e) & = & \lambda y'.[e_2/x]([y'/y]e) \quad (\text{ただし } y' \text{ は新しい変数とする}) \\
[e_2/x](ee') & = & ([e_2/x]e)([e_2/x]e')
\end{array}$$

1.5 Objective Caml による実装

以上の定義にもとづいて、λ計算のインタプリタを Objective Caml で実装してみよう。

```
type exp = (* λ式を表すデータ型 *)
    Var of string       (* 変数 *) (* 変数名は文字列であらわすことにする *)
  | Abs of string * exp (* λ抽象 *)
  | App of exp * exp    (* 関数適用 *)

let gensym = (* 新しい変数名を作る補助的な関数 *)
  let counter = ref 0 in          (* 整数0を持った参照セルcounterを作る *)
  fun () ->                       (* ダミーの引数()を受け取ったら… *)
    incr counter;                 (* counterの中身を一つ増やす *)
    "g" ^ string_of_int !counter (* g1, g2, g3, ...等の文字列を返す *)

let rec subst e2 x e1 = (* [e2/x]e1を求めて返す *)
  match e1 with
    Var(y) ->
      if x = y then e2 else Var(y)
  | Abs(y, e) ->
      let y' = gensym () in
      Abs(y', subst e2 x (subst (Var(y')) y e))
  | App(e, e') ->
      App(subst e2 x e, subst e2 x e')

let rec step e = (* eを受け取り、e -> e'となるe'のリストを返す *)
  match e with
    Var(x) -> []
  | Abs(x, e0) ->
      (* (R-Abs)より *)
      List.map
        (fun e0' -> Abs(x, e0'))
        (step e0)
```

```
  | App(e1, e2) ->
      (* (R-Beta) より *)
      (match e1 with
        Abs(x, e0) -> [subst e2 x e0]
      | _ -> []) @
      (* (R-App1) より *)
      List.map
        (fun e1' -> App(e1', e2))
        (step e1) @
      (* (R-App2) より *)
      List.map
        (fun e2' -> App(e1, e2'))
        (step e2)

let rec repeat e = (* 簡約できなくなるまで step を反復する *)
  match step e with
    [] -> e
  | e' :: _ -> repeat e'
```

例:

```
# let e1 = Abs("x", Var("x")) ;; (* (λ x. x) *)
val e1 : exp = Abs ("x", Var "x")
# step e1 ;;
- : exp list = []
# let e2 = App(e1, e1) ;; (* (λ x. x) (λ x. x) *)
val e2 : exp = App (Abs ("x", Var "x"), Abs ("x", Var "x"))
# step e2 ;;
- : exp list = [Abs ("x", Var "x")]
# let e3 = App(e2, e2) ;; (* ((λ x. x) (λ x. x)) ((λ x. x) (λ x. x)) *)
val e3 : exp =
  App (App (Abs ("x", Var "x"), Abs ("x", Var "x")),
   App (Abs ("x", Var "x"), Abs ("x", Var "x")))
```

```
# step e3 ;;
- : exp list =
[App (Abs ("x", Var "x"), App (Abs ("x", Var "x"), Abs ("x", Var "x")));
 App (App (Abs ("x", Var "x"), Abs ("x", Var "x")), Abs ("x", Var "x"))]
# repeat e3 ;;
- : exp = Abs ("x", Var "x")
```

問 3. 他の例を作って試してみよ。特に、`step e` が長さ 2 以上のリストを返す場合について、`repeat e` の結果を考察せよ。 □

1.6 簡約結果の一意性、合流性、簡約戦略

先の例からもわかるように、一つの λ 式 e に対して、$e \to e'$ となる e' が一つとは限らない。つまり、0 個かもしれないし、2 個以上かもしれない。

0 個の場合は、それ以上の簡約ができない、つまり計算が停止したことになる。ただし、計算の停止といっても、エラーによる異常終了かもしれないし、正常終了かもしれない。どちらであるかは、簡約結果となる λ 式の形により定めることが多い。たとえば「関数適用 e_1e_2 の形で終わってしまったら、計算が中途半端に停止したとみなしてエラー、その他の形ならば OK」などである。

2 個以上の場合は、複数の簡約が可能である、つまり計算の**方法**が一意でないことになる。では、計算の**結果**も一意ではなくなってしまうのだろうか？　実は、λ 計算では

計算の方法は一意でないが、結果は一意である

ことがわかっている。より正確に記述すると、次の通りである。

定理 1 (簡約結果の一意性). 任意の λ 式 e について、もし $e \to^* e_1 \not\to$ かつ $e \to^* e_2 \not\to$ ならば、$e_1 = e_2$ である。 □

ただし、$\to^*$ は $\to$ の反射的推移的閉包、つまり $\to$ を 0 回以上合成した関係である。すなわち、一般に $e_0 \to^* e_n$ とは、$e_0 \to e_1 \to e_2 \to \ldots \to e_{n-1} \to e_n$ $(n \geq 0)$ となる $e_1, e_2, \ldots, e_{n-1}$ が存在することである。あるいは

$$\frac{}{e \to^* e}$$

$$\frac{e_1 \to e_2 \qquad e_2 \to^* e_3}{e_1 \to^* e_3}$$

という規則により定義することもできる。また、$e \not\to$ とは、$e \to e'$ となる e' が存在しないことである。

上の定理は、次の性質から導かれる。

定理 2 (合流性). 任意の λ 式 e について、もし $e \to^* e_1$ かつ $e \to^* e_2$ ならば、$e_1 \to^* e'$ かつ $e_2 \to^* e'$ となる e' が存在する。 □

問 4. 定理 2 を仮定して、定理 1 を証明せよ。 □

簡約結果の一意性や合流性は、**もし簡約が停止すれば**結果は一意であると言っているに過ぎず、絶対に簡約が停止するとは主張していない。実際に、たとえば

$$\Omega \quad = \quad (\lambda x.xx)(\lambda x.xx)$$

とおくと、

$$\Omega \quad \to \quad \Omega \quad \to \quad \Omega \quad \to \quad \ldots$$

という、無限に停止しない簡約しか存在しない。

さらに、$(\lambda x.y)\Omega$ のような式を考えると、

$$(\lambda x.y)\Omega \quad \to \quad y$$

のように停止する簡約と、

$$(\lambda x.y)\Omega \quad \to \quad (\lambda x.y)\Omega \quad \to \quad (\lambda x.y)\Omega \quad \to \quad \ldots$$

のように停止しない簡約の両方が存在する。

そこで一般に任意の λ 式について、できるだけ簡約が停止するような**簡約戦略**があるとうれしい。つまり、「ある簡約が停止するならば、X にしたがった簡約も停止する」というような X がほしい。そのような X としては「できるだけ外側および左側から簡約していく」**最外最左簡約**がある。たとえば $(\lambda x.y)\Omega$ なら、引数 Ω よりも $(\lambda x.y)\Omega$ 全体のほうが外側にあるので、後者から簡約する。最外最左簡約が停止しないような λ 式は、どのような簡約も停止しないことがわかっている。このことから、最外最左簡約は**正規順序簡約**ともいわれる。

先のインタプリタは、関数 `repeat` において `step e` の第一要素しか使っていないが、`step` の実装をよく見ると、実は最外最左簡約になっている。

問 5. 先のインタプリタを、最外最左簡約ではなくなるように変えてみよ。また、その違いがわかるような例を示せ。 □

ちなみに ML は関数適用において、まず引数を評価するので正規順序簡約**ではない**。実際に、

```
# let rec loop x = loop x ;; (* 停止しない関数 *)
val loop : 'a -> 'b = <fun>
# let y = (fun z -> z) ;; (* y を適当に定義する *)
val y : 'a -> 'a = <fun>
# (fun x -> y) (loop ()) ;;
Interrupted.
```

のように、停止しない引数に $\lambda x.y$ を適用しても停止しない。このように、まず引数を簡約してから関数を適用する簡略戦略を**値呼び** *(call-by-value)* という。逆に、引数を簡約しないで関数を適用する簡略戦略を**名前呼び** *(call-by-name)* という。名前呼び簡約を実現している言語としては、Haskell がある。ただし、実装では**必要呼び**(call-by-need) という、より効率のよい方法を使用している。

1.7 λ 式による様々なデータの表現

先に「およそコンピュータで可能な計算は、λ 計算でも表現できる」と述べた。したがって、そもそも「およそコンピュータで扱えるデータは、λ 式でも表せる」はずである。それらの例をいくつか見ていこう。

1.7.1 論理値

論理値 `true`, `false` および条件式 `if` e_1 `then` e_2 `else` e_3 は、

$$\begin{aligned} \texttt{true} &= \lambda t.\lambda f.t \\ \texttt{false} &= \lambda t.\lambda f.f \\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 &= e_1 e_2 e_3 \end{aligned}$$

と表せる。なぜならば、

$$\begin{aligned} \texttt{if true then}\ e_2\ \texttt{else}\ e_3 &= \texttt{true}\, e_2 e_3 \\ &= (\lambda t.\lambda f.t) e_2 e_3 \\ &\to (\lambda f.e_2) e_3 \\ &\to e_2 \end{aligned}$$

かつ

$$\begin{aligned}\texttt{if false then } e_2 \texttt{ else } e_3 &= \texttt{false}\, e_2 e_3 \\ &= (\lambda t.\lambda f.f) e_2 e_3 \\ &\to (\lambda f.f) e_3 \\ &\to e_3\end{aligned}$$

だからである。

より直感的には、次のように考えることもできる。一般に論理値 b についての、もっとも基本的な演算は条件分岐 `if` b `then` e_2 `else` e_3 である。そこで、この条件分岐の機能を b 自身にもたせる。つまり、`then` 節 e_2 と `else` 節 e_3 を受け取って、そのどちらかを返すような関数により、b を表すことにする。すると、`true` は e_2 と e_3 を受け取って、e_2 のほうを返す関数

$$\lambda t.\lambda f.t$$

で表せる。同じように、`false` は e_2 と e_3 を受け取って、e_3 のほうを返す関数

$$\lambda t.\lambda f.f$$

で表せる。b 自体に条件分岐の機能をもたせたので、`if` 式は単に b に `then` 節と `else` 節を渡すだけで良い。

1.7.2 組

組は

$$\begin{aligned}(e_1, e_2) &= \lambda c.c e_1 e_2 \\ \texttt{match } e_1 \texttt{ with } (f, s) \texttt{ -> } e_2 &= e_1 (\lambda f.\lambda s.e_2)\end{aligned}$$

のように表せる。

直感的には論理値の場合と同様で、組に対する基本演算であるパターンマッチングの機能を、組そのものに持たせている。つまり、「パターンマッチングの節を表す関数 c を与えられたら、それを第一要素 e_1 と第二要素 e_2 に適用する」という関数 $\lambda c.c e_1 e_2$ により、組 (e_1, e_2) を表している。すると `match` 式は単に、組（を表す関数）をパターンマッチングの節（を表す関数）に適用するだけでよい。

問 6. 次の簡約が導けることを確認せよ。

$$\texttt{match}\ (e_1, e_2)\ \texttt{with}\ (x, y)\ \texttt{->}\ e_3 \quad \rightarrow^* \quad [e_2/y][e_1/x]e_3$$

□

1.7.3 自然数

自然数は

$$\begin{aligned} 0 &= \lambda s.\lambda z.z \\ 1 &= \lambda s.\lambda z.sz \\ 2 &= \lambda s.\lambda z.s(sz) \\ 3 &= \lambda s.\lambda z.s(s(sz)) \\ &\vdots \end{aligned}$$

と表せる。このような λ 式を**チャーチ数**ないし自然数の**チャーチ符号化**(Church encoding)という。

なぜこのように表すのだろうか？　一般に自然数についての、もっとも基本的な演算はループである。ループの初期値を z、一回ごとに適用する関数を s とすると

- 自然数 0 に対するループの値は z
- 1 に対する値は $s(z)$
- 2 に対する値は $s(s(z))$
- 3 に対する値は $s(s(s(z)))$
- 以下同様

となる。s や z の値はループによって異なるので、どのような s と z にも適用できるように λ 抽象して、

$$\begin{aligned} 0 &= \lambda s.\lambda z.z \\ 1 &= \lambda s.\lambda z.sz \\ 2 &= \lambda s.\lambda z.s(sz) \\ 3 &= \lambda s.\lambda z.s(s(sz)) \\ &\vdots \end{aligned}$$

となる。

こうすると、たとえば足し算 $m+n$ は

$$m+n \quad = \quad \lambda s.\lambda z.ns(msz)$$

と定義できる。これは次のように考えられる。まず式 msz により、初期値 z に対して、ループ s を m 回繰り返す。その msz を初期値として s を n 回繰り返せば、合わせて $m+n$ 回 s を繰り返したことになるはずである。実際に簡約してみると、たとえば

$$\begin{aligned}
1+2 &= \lambda s.\lambda z.1s(2sz) \\
&= \lambda s.\lambda z.(\lambda s.\lambda z.sz)s(2sz) \\
&\to \lambda s.\lambda z.(\lambda z.sz)(2sz) \\
&\to \lambda s.\lambda z.s(2sz) \\
&= \lambda s.\lambda z.s((\lambda s.\lambda z.s(sz))sz) \\
&\to \lambda s.\lambda z.s((\lambda z.s(sz))z) \\
&\to \lambda s.\lambda z.s(s(sz)) \\
&= 3
\end{aligned}$$

となるし、先のインタプリタで試しても

```
# let one =
    Abs("s",
      Abs("z",
        App(Var("s"), Var("z")))) ;;
val one : exp = Abs ("s", Abs ("z", App (Var "s", Var "z")))
# let two =
    Abs("s",
      Abs("z",
        App(Var("s"), App(Var("s"), Var("z"))))));;
val two : exp = Abs ("s", Abs ("z", App (Var "s", App (Var "s", Var "z"))))
# let plus m n =
    Abs ("s",
      Abs ("z",
```

```
          App (App (m, Var("s")),
               App(App(n, Var("s")), Var("z"))))) ;;
val plus : exp -> exp -> exp = <fun>
# repeat (plus one two) ;;
- : exp =
Abs ("s", Abs ("z", App (Var "s", App (Var "s", App (Var "s", Var "z")))))
```

となる。

掛け算 $m \times n$ は

$$m \times n \quad = \quad \lambda s.\lambda z.n(\lambda z'.msz')z$$

と定義できる。これは次のように考えられる。「s を m 回繰り返す関数」を n 回繰り返せば、s を $m \times n$ 回繰り返したことになるはずである。この「s を m 回繰り返す関数」が $\lambda z'.msz'$ であり、それを「n 回繰り返す」のが $n(\lambda z'.msz')z$ である。ループが二重になっているので、外側のループの初期値 z は全体の初期値と同じだが、内側のループの初期値 z' は呼び出されるごとに変わっていくことに気をつけよ。

問 7. $1+2 \rightarrow^* 3$ を確かめたのと同じように、$2 \times 3 \rightarrow^* 6$ となることを、手動・自動（インタプリタ）の両方で確認せよ。 □

問 8. $m+n$ や $m \times n$ を λ 式で表したように、$m-n$ を λ 式で表せ。ただし $m < n$ ならば $m-n=0$ とする。（この問は難しい。もし自力でできたら住井にメールせよ。） □

1.7.4 再帰関数

再帰関数 $g(y) = e$ は、

$$\mathtt{fix}_f \quad = \quad (\lambda x.f(xx))(\lambda x.f(xx))$$

という λ 式を使って

$$g \quad = \quad \mathtt{fix}_{\lambda g.\lambda y.e}$$

と表せる[3]。その理由を説明する。

[3] e の中に g が含まれているかもしれないので、$g = \lambda y.e$ では定義になっていないことに注意しよう。

まず、任意の λ 式 f について、$\mathtt{fix}_f$ は

$$\begin{aligned}\mathtt{fix}_f &= (\lambda x.f(xx))(\lambda x.f(xx)) \\ &\to f((\lambda x.f(xx))(\lambda x.f(xx))) \\ &= f(\mathtt{fix}_f)\end{aligned}$$

と簡約できる。つまり、$\mathtt{fix}_f$ は f の**不動点**になっている。

すると、$g(y)$ は

$$\begin{aligned}g(y) &= (\mathtt{fix}_{\lambda g.\lambda y.e})y \\ &\to (\lambda g.\lambda y.e)(\mathtt{fix}_{\lambda g.\lambda y.e})y \\ &= (\lambda g.\lambda y.e)gy \\ &\to (\lambda y.e)y \\ &\to e\end{aligned}$$

と簡約される[4]。よって、$g(y) = e$ という定義が実現されている。

2 単純型つき λ 計算

前節の λ 計算には型の概念がまったくなかったので、たとえば論理値の $\mathtt{false}$ に対し自然数の 2 を足す

$$\mathtt{false} + 2$$

等という意味のない演算を行っても、

$$\begin{aligned}\mathtt{false} + 2 &= (\lambda t.\lambda f.f) + 2 \\ &= (\lambda s.\lambda z.z) + 2 \\ &= 0 + 2 \\ &\to^* 2\end{aligned}$$

などと簡約できてしまう。このような混乱があってはアセンブリ言語と大差がなく危険である。

[4] ここでは、自由な g は $g = \mathtt{fix}_{\lambda g.\lambda y.e}$ と定義されたメタな変数であり、束縛された g は通常の λ 計算の変数である。引っかかる人は、$\mathtt{fix}_{\lambda g.\lambda y.e}$ のかわりに、α 同値な $\mathtt{fix}_{\lambda g'.\lambda y.[g'/g]e}$ で確認せよ。

また、通常のプログラミング言語のように、自然数や論理値などを λ 式ではなくプリミティブな定数として導入しても、もし型がなかったら

$$\texttt{if}\ 3+7\ \texttt{then}\ 10\ \texttt{else}\ \texttt{false}$$

のようにナンセンスな式を書くことができてしまう。

そこで、式の種類すなわち型を区別することにより、プログラムの実行（λ 式の簡約）以前に、そのような混乱を防止する方法について考察する。

2.1 型

まず、型 τ の構文を

$$\begin{array}{rcll} \tau\ \text{(型)} & ::= & b & \text{(基本型)} \\ & | & \tau_1 \to \tau_2 & \text{(関数型)} \end{array}$$

と定義する。ただし基本型というのは、論理値の型 bool や自然数の型 nat など、いわば「型の定数」である。もし基本型がなかったら、

$$\tau\ \text{(型)}\ ::=\ \tau_1 \to \tau_2\ \text{(関数型)}$$

と定義される型 τ の集合は空になってしまうことに注意しよう。

2.2 型判定

型 τ の構文を定義したので、次に「式 e が型 τ を持つ」とはどういうことか考えていく。

たとえば、$\lambda x.x$ という式を考える。この式は、引数 x を与えられたら、その x を結果として返す、という関数である。したがって、型 τ の引数を与えられたら、返値の型も τ となる。よって、

$$\text{式}\ \lambda x.x\ \text{は型}\ \tau \to \tau\ \text{を持つ}$$

といえるはずである。これを

$$\vdash \lambda x.x : \tau \to \tau$$

と書く。一般に

$$\text{式}\ e\ \text{が型}\ \tau\ \text{を持つ}$$

という意味の命題を**型判定**といい、

$$\vdash \quad e : \tau$$

と表記する。

では、$\lambda x.y$ という式はどうだろうか。この式は先の式と違い、自由変数 y を持つ。一般に自由変数の値は式の外から与えられるので、その型も式の外から与えられるはずである。そこで、

y が型 τ を持つならば、式 $\lambda x.y$ は型 $\tau' \to \tau$ を持つ

という型判定

$$y : \tau \quad \vdash \quad \lambda x.y : \tau' \to \tau$$

が考えられる。一般に、

$x_1, x_2, \ldots, x_n$ がそれぞれ型 $\tau_1, \tau_2, \ldots, \tau_n$ を持つならば、式 e は型 τ を持つ

という型判定を

$$x_1 : \tau_1,\ x_2 : \tau_2,\ \ldots,\ x_n : \tau_n \quad \vdash \quad e : \tau$$

と書く。たとえば、

$$m : \mathtt{nat},\ n : \mathtt{nat},\ \mathtt{plus} : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat} \quad \vdash \quad \mathtt{plus}\, m\, n : \mathtt{nat}$$

という型判定が成り立つはずである。ただし、$\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$ とは $\mathtt{nat} \to (\mathtt{nat} \to \mathtt{nat})$ のことである。より一般に、$\to$ は**右結合的**である。

型判定において、変数に型を与える $x_1 : \tau_1,\ x_2 : \tau_2,\ \ldots,\ x_n : \tau_n$ という部分のことを**型環境**といい、Γ や Δ などのメタ変数で表記する。型環境は、変数 $x_1, x_2, \ldots, x_n$ から型 $\tau_1, \tau_2, \ldots, \tau_n$ への写像とみなすことができる。

2.3 型つけ規則

前節では型判定の表記と直感的な意味について説明したが、それだけでは

どのような Γ, e, τ に対して、$\Gamma \vdash e : \tau$ という型判定が成り立つのか

まだわからない。これをきちんと定義するのが**型つけ規則**である。

先に定めた型による、純粋な λ 式に対する型つけ規則は、次のように与えられる。

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau} \quad \text{(T-App)}$$

規則 (T-Var) は、型環境 Γ に $x : \tau$ が含まれていたら、Γ の下で変数 x は型 τ を持つ、と述べている。規則 (T-App) は、型環境 Γ の下で、式 e_1 が関数型 $\tau' \to \tau$ を持ち、式 e_2 が引数の型 τ' を持つならば、関数適用 $e_1 e_2$ は結果の型 τ を持つ、と述べている。

規則 (T-Abs) の意味は次の通りである。型環境 Γ の下で、関数 $\lambda x.e$ が型 $\tau_1 \to \tau_2$ を持つためには、Γ に引数の型 $x : \tau_1$ をつけくわえた型環境 $\Gamma, x : \tau_1$ の下で、関数の本体 e が型 τ_2 を持っていれば良い。これは「x として型 τ_1 の引数を与えたら、e は型 τ_2 の結果を返す」という直感を反映している。

例:

$$\dfrac{\dfrac{\dfrac{}{s : \mathtt{nat} \to \mathtt{nat}, z : \mathtt{nat} \vdash s : \mathtt{nat} \to \mathtt{nat}}\text{T-Var} \quad \dfrac{}{s : \mathtt{nat} \to \mathtt{nat}, z : \mathtt{nat} \vdash z : \mathtt{nat}}\text{T-Var}}{s : \mathtt{nat} \to \mathtt{nat}, z : \mathtt{nat} \vdash sz : \mathtt{nat}}\text{T-App}}{s : \mathtt{nat} \to \mathtt{nat} \vdash \lambda z.sz : \mathtt{nat} \to \mathtt{nat}}\text{T-Abs}}{\vdash \lambda s.\lambda z.sz : (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat} \to \mathtt{nat}}\text{T-Abs}$$

2.4 安全性

そもそも型は、$\mathtt{false} + 5$ や $\mathtt{if}\ 3 + 7\ \mathtt{then}\ 10\ \mathtt{else}\ \mathtt{false}$ といったナンセンスなプログラムを拒否するために導入したのであった。では、前節の規則により型つけできたプログラムは、本当にエラーにならず、ちゃんと実行できるのだろうか。

そのような性質を証明するためには、まず「エラーとは何か」ということを定めなければいけない。λ 計算では、プログラムは λ 式として表され、その簡約がプログラムの実行に相当するのであった。したがって、簡約が停止したときに、その λ 式がちゃんと何らかの**値**に

なっていれば正常終了、そうでなければエラーとみなすことができる。さらに純粋なλ計算では、すべての値はλ抽象すなわち関数として表されるので、「値 = 関数」すなわち

$$v \quad (\text{値}) \quad ::= \quad \lambda x.e \quad (\text{関数})$$

と定義できる（もし自然数や論理値などをプリミティブな定数として追加したら、それらも値に含まれる）。

すると、以下の定理が成立する。つまり、前節の規則により型つけできるλ式は、決してエラーを起こさない、といえる。

定理 3 (安全性). もし$\vdash e : \tau$かつ$e \to^* e' \not\to$ならば、e'は値である。 □

この定理は、以下の二つの補題から導かれる。

補題 1 (進行). もし$\vdash e : \tau$ならば、eは値であるか、あるいは$e \to e'$となるe'が存在する。 □

補題 2 (型の保存). もし$\vdash e : \tau$かつ$e \to e'$ならば、$\vdash e' : \tau$である。 □

どちらも証明は$\vdash e : \tau$の導出についての帰納法による。ただし後者の証明の途中で、以下の補題が必要となる。

補題 3 (代入補題). もし$\Gamma \vdash e_1 : \tau_1$かつ$\Gamma, x : \tau_1 \vdash e_2 : \tau_2$ならば、$\Gamma \vdash [e_1/x]e_2 : \tau_2$である。 □

証明. $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$の導出についての帰納法による。 □

問 9. 補題1と補題2を利用して、定理3を証明せよ。 □

2.5 データ抽象

1.7.1節で、論理値および条件式は

$$\begin{aligned} \texttt{true} &= \lambda t.\lambda f.t \\ \texttt{false} &= \lambda t.\lambda f.f \\ \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 &= e_1 e_2 e_3 \end{aligned}$$

のようにλ式で表せると述べた。しかし、論理値・条件式の表し方はこれだけではない。たとえば、

$$\begin{aligned} \texttt{true} &= \lambda f.\lambda t.t \\ \texttt{false} &= \lambda f.\lambda t.f \\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 &= e_1 e_3 e_2 \end{aligned}$$

のように、true と false を逆にしても同じのはずである。したがって、たとえば

$$\texttt{true} : \texttt{bool}, \texttt{false} : \texttt{bool}, \texttt{if} : \texttt{bool} \to \texttt{nat} \to \texttt{nat} \to \texttt{nat} \ \vdash \ e : \texttt{nat}$$

のように true, false, if を使用する（純粋な）λ式 e があったら、

$$[\lambda t.\lambda f.t/\texttt{true}][\lambda t.\lambda f.f/\texttt{false}][\lambda b.\lambda t.\lambda f.btf/\texttt{if}]e$$

のように1番目の実装を代入しても、

$$[\lambda f.\lambda t.t/\texttt{true}][\lambda f.\lambda t.f/\texttt{false}][\lambda b.\lambda t.\lambda f.bft/\texttt{if}]e$$

のように2番目の実装を代入しても、簡約結果の値は変わらないはずである。

このように、型を用いると単に「エラーが起きない」というだけでなく、

データ構造の実装を変更しても、プログラム全体の結果は変化しない

という性質を保証することができる。このような手法のことを**データ抽象**という。

データ抽象のポイントは、「データ構造を使用するプログラムが、その具体的な表され方を知らず、あくまで抽象的な型の下で扱う」ということにある。たとえば上の例でも、もしプログラム e が true を関数として適用したら、2つの実装を区別することができてしまうので、e の値が変わらないとはいえなくなる。しかし規則 (T-App) より、そのような関数適用は型つけできないので、ありえない。

Objective Caml にも、様々な抽象データ構造のライブラリがある。たとえば、値から値への対応を記憶するハッシュテーブルなどである。

```
# let ht = Hashtbl.create 10 ;; (* 新しいハッシュテーブル ht を作る *)
val ht : ('_a, '_b) Hashtbl.t = <abstr>
# ht ;; (* ht の値は抽象化されており見ることができない *)
- : (string, int) Hashtbl.t = <abstr>
```

```
# Hashtbl.add ht "abc" 123 ;; (* 値"abc"から値123への対応をhtに追加する *)
- : unit = ()
# Hashtbl.add ht "de" 456 ;; (* "de"から456への対応を追加する *)
- : unit = ()
# Hashtbl.add ht "f" 789 ;; (* "f"から789への対応を追加する *)
- : unit = ()
# Hashtbl.find ht "de" ;; (* "de"に対応する値を検索する *)
- : int = 456
# Hashtbl.find ht "g" ;; (* "g"に対応する値は存在しないので例外が発生する *)
Exception: Not_found.
# ht + 1 ;; (* Hashtblモジュールで定義された演算以外はhtに適用できない *)
Characters 0-2:
  ht + 1 ;; (* Hashtblモジュールで定義された演算以外はhtに適用できない *)
  ^^
This expression has type (string, int) Hashtbl.t but is here used with type
  int
```

さらに勉強するには

λ計算に関する日本語の本は限られているが、たとえば

- プログラミング言語の基礎理論 情報数学講座9, 大堀 淳, 共立出版, ISBN 4-320-02659-4.

が良い。英語で良ければ[5]いろいろとあるが、

- Types and Programming Languages. Benjamin C. Pierce. The MIT Press. ISBN 0-262-16209-1.

がわかりやすい。

[5]大学でも企業でも、技術書・専門書ぐらい英語で容易に読めないと困るはずだが…