第 5 章: 型無し λ 計算

June 2, 2025

1. 前回までの内容
    1. 型無し λ 計算の抽象構文
    2. 変数の出現
    3. 評価戦略 (完全 β 簡約，正規評価順序)

今回は 5 章の残りをする．ただし操作的意味論を明確にするため，5.3 章の内容を先に扱う．

名前呼び

名前呼びは，informal には抽象内部に簡約基を持たない (i.e. 弱冠頭正規形) 項のみに制限された正規評価順序である．変種が Haskell などで使われている．

名前呼び

名前呼び戦略とは，最左・最外の簡約基から簡約し，抽象内部の簡約を許さない評価戦略である．

名前呼び戦略のもとでの簡約列の例．

$$\underline{\mathtt{id}(\mathtt{id}(\lambda z.\ \mathtt{id}\ z))} \to \underline{\mathtt{id}(\lambda z.\mathtt{id}\ z)} \twoheadrightarrow \lambda z.\mathtt{id}\ z \twoheadrightarrow$$

正規評価順序で行なえていた抽象内部の簡約が行なえないので，最後に得られた項が $\lambda z.z$ に簡約されることはない．

名前呼び

具体的に操作的意味論を定める．関数部が正規形になった時に，(引数が値か否に関わらず)β 簡約を施すことに注意すると

名前呼びの操作的意味論

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}\ \text{(E-APP1)}$$

$$\frac{}{(\lambda x.t_{12})t_2 \to [t_2 \mapsto x]\, t_{12}}\ \text{(E-APPABS)}$$

値は λ 抽象である．次に見る値呼びの評価規則とは異なり，引数に関する評価関係は定められていないことに注意する．

値呼び

値呼び

簡約基の右側がこれ以上簡約できない項，つまり値に評価されているときだけ最外簡約基を簡約する評価戦略を値呼びという．

$\mathtt{id}(\mathtt{id}(\lambda \mathtt{z}.\mathtt{id}\ \mathtt{z}))$ の値呼びでの簡約列．

$$\mathtt{id}(\underline{\mathtt{id}(\lambda \mathtt{z}.\mathtt{id}\ \mathtt{z})}) \to \underline{\mathtt{id}\ (\lambda \mathtt{z}.\mathtt{id}\ \mathtt{z})} \to \lambda \mathtt{z}.\mathtt{id}\ \mathtt{z}$$

値呼びは正格である．つまり，函数本体で使用されるかにかかわらず引数は常に評価される．これは，引数を函数本体で必要なときだけ評価する名前呼びや必要呼び戦略と対照的である．

値呼びの操作的意味論

まず，値呼びにおける構文要素を定める．

値呼びにおける構文要素

項は以下の BNF で定まる:

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

値は λ 抽象値である．したがって，以下の BNF で定まる:

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

次に評価関係を定めよう．値呼びで簡約が許されるのは最も外側の redex である．そうでない項は，はじめ左側，次いで右側の部分項が値となるまで簡約されるまで簡約される．定式化すると次のようになる．

値呼びの操作的意味論

値呼びにおける評価関係

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}\ \text{(E-APP1)}$$

$$\frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'}\ \text{(E-APP2)}$$

$$\frac{}{(\lambda x.t_{12})\ v_2 \to [x \mapsto v_2]\ t_{12}}\ \text{(E-APPABS)}$$

E-APP1 により，関数適用は関数部分が値となるまで評価されたのち，E-APP2 によって左辺が値となるまで評価が進む．最後に E-APPABS により右辺が値となった redex のみ評価される．

λ 計算でのプログラミング

1. 複数の引数: カリー化
   - 複数個の引数を取る関数は高階関数をネストして作れる (カリー化).
   - たとえば，項 $\mathtt{f} = \lambda(\mathtt{x}, \mathtt{y}).\mathtt{s}$(ただし $\mathtt{x}, \mathtt{y} \in FV(\mathtt{s})$) は $\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{s}$ とかける.
     - x に対して値 v が代入されたとき，各 y に値 w が代入されると $[\mathtt{y} \mapsto \mathtt{w}]\,[\mathtt{x} \mapsto \mathtt{v}]\,\mathtt{s}$ を返す関数
   - カリー化された関数の簡約列の例:
     $(((\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{s})\mathtt{v})\mathtt{w}) \longrightarrow ((\lambda\mathtt{y}.\,[\mathtt{x} \mapsto \mathtt{v}]\,\mathtt{s})\mathtt{w}) \longrightarrow [\mathtt{y} \mapsto \mathtt{w}]\,[\mathtt{x} \mapsto \mathtt{v}]\,\mathtt{s}$
2. ブール値
   - 項 $\mathtt{tru}, \mathtt{fls}$ を $\mathtt{tru} = \lambda\mathtt{t}.\lambda\mathtt{f}.\mathtt{t}, \mathtt{fls} = \lambda\mathtt{t}.\lambda\mathtt{f}.\mathtt{f}$ と定める.
   - if 演算子に相当する test コンビネータを $\mathtt{test} = \lambda\mathtt{l}.\lambda\mathtt{m}.\lambda\mathtt{n}.\mathtt{l}\ \mathtt{m}\ \mathtt{n};$ とする.
     - $\mathtt{tru}, \mathtt{fls}$ は二引数のうちそれぞれ前者と後者を返すので，test について真理値同様に振る舞う．i.e. 真理値を表現している.
   - 他の論理演算子も test で書ける．たとえば論理和は $\mathtt{and} = \lambda\mathtt{b}.\lambda\mathtt{c}.\mathtt{test}\ \mathtt{b}\ \mathtt{c}\ \mathtt{fls}.$
3. 二つ組
   - 二つ組のコンストラクタ pair と射影関数 $\mathtt{fst}, \mathtt{snd}$ を次のように定義する:$\mathtt{pair} = \lambda\mathtt{f}.\lambda\mathtt{s}\lambda\mathtt{b}.\mathtt{b}\ \mathtt{f}\ \mathtt{s};\ \mathtt{fst} = \lambda\mathtt{p}.\mathtt{p}\ \mathtt{tru};\ \mathtt{snd} = \lambda\mathtt{p}.\mathtt{p}\ \mathtt{fls}$
   - これらは順序対として振る舞う．つまり任意の項 $\mathtt{v}, \mathtt{w}$ について $\mathtt{v} = \mathtt{fst}(\mathtt{pair}\ \mathtt{v}\ \mathtt{w}), \mathtt{w} = \mathtt{snd}(\mathtt{pair}\ \mathtt{v}\ \mathtt{w}).$

λ 計算でのプログラミング: Church 数

型無し算術式では，数は定数 0 と後者関数 succ で帰納的に定義された．λ 計算でも同様であり，とくにそれは Church 数と呼ばれる．

Church 数

各自然数 n について，Church 数 c_n を次のように定義する．

$$\mathsf{c}_n = \lambda \mathsf{s}.\lambda \mathsf{z}.\overbrace{(\mathsf{s} \ldots (\mathsf{s}}^{n\text{ 個}}\ \mathsf{z}) \ldots)$$

とくに $\mathsf{c}_0 = \lambda \mathsf{s}.\lambda \mathsf{z}.\mathsf{z}$ である．

Church 数 c_n は，後者関数 s とゼロ z を取り z に s を n 回適用する．

後者関数 scc は，各 c_n に対して，c_n に後者関数とゼロを適用し，更に後者関数を適用する関数としてかける．つまり，

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

以降のスライドでは，Church 数の加算，減算，乗算およびゼロ値判定を定める．

λ 計算でのプログラミング: Church 数

1. 加算関数は，各 $\mathtt{m}, \mathtt{n}$ に対して $\mathtt{c}_\mathtt{n}$ に $\mathtt{s}$ を n 回適用する関数として書ける:

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

2. 乗算関数は，各 $\mathtt{m}, \mathtt{n}$ に対して $\mathtt{plus}\ \mathtt{n}$ を m 回だけ $\mathtt{c}_0$ に適用する関数として書ける．Church 数 $\mathtt{c}_\mathtt{n}$ が乗算関数 $\mathtt{s}$ を項 $\mathtt{z}$ に n 回適用することを用いた:

$$\mathtt{times} = \lambda \mathtt{m}.\lambda \mathtt{n}.\mathtt{m}\ (\mathtt{plus}\ \mathtt{n})\ \mathtt{c}_0 = \lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{z}.\mathtt{m}(\mathtt{nz})$$

   - cf. 乗算の単位元は 1 だから，各 $\mathtt{m}, \mathtt{n}$ に対して $\mathtt{c}_{\mathtt{m}^\mathtt{n}}$ を返す関数 $\mathtt{exp}$ は

$$\mathtt{exp} = \lambda \mathtt{m}.\lambda \mathtt{n}.\mathtt{m}\ (\mathtt{times}\ \mathtt{n})\ \mathtt{c}_1$$

3. ゼロ値判定のための関数は，各 $\mathtt{n}$ に対して Church 数 $\mathtt{c}_\mathtt{n}$ が $\mathtt{c}_0$ のとき $\mathtt{tru}$，それ以外は常に $\mathtt{fls}$ を返す関数として書ける:

$$\mathtt{iszro} = \lambda \mathtt{m}.\mathtt{m}(\lambda \mathtt{x}.\mathtt{fls})\ \mathtt{tru}$$

減算関数を定めるために前者関数 prd を用意する．pred はメモ化を使うと書ける:

前者関数

$\mathtt{prd} = \lambda \mathtt{m}.\mathtt{fst}(\mathtt{m}\ \mathtt{ss}\ \mathtt{zz})$．ただし，

- $\mathtt{zz} = \mathtt{pair}\ \mathtt{c}_0\ \mathtt{c}_0$
- $\mathtt{ss} = \lambda \mathtt{p}.\mathtt{pair}(\mathtt{snd}\ \mathtt{p})\ (\mathtt{plus}\ \mathtt{c}_1\ (\mathtt{snd}\ \mathtt{p}))$

- ss は二つ組 $\mathtt{pair}\ \mathtt{c}_i\ \mathtt{c}_j$ を取り $\mathtt{pair}\ \mathtt{c}_j\ \mathtt{c}_{j+1}$ を返すから，$m = 0$ のとき項 $\mathtt{m}\ \mathtt{ss}\ \mathtt{zz}$ は $\mathtt{pair}\ \mathtt{c}_0\ \mathtt{c}_0$，それ以外は帰納法により $\mathtt{pair}\ \mathtt{c}_{m-1}\ \mathtt{c}_m$.
- 長さ n の引数が与えられた関数 f の評価回数を $T(f)(n)$ と書くと，$T(\mathtt{prd})(n) = \mathcal{O}(n)$．（次スライド）

prd により，減算関数 sub は $\mathtt{sub} = \lambda \mathtt{m}.\lambda \mathtt{n}.\mathtt{m}\ \mathtt{prd}\ \mathtt{n}$ と書ける．

$\mathtt{prd}(\mathtt{c}_n)$ の簡約列を考える.

$$\mathtt{prd}(\mathtt{c}_n) = (\lambda \mathtt{m}.\mathtt{fst}(\mathtt{m}\ \mathtt{ss}\ \mathtt{zz}))(\mathtt{c}_n)$$

$$\to \mathtt{fst}(\mathtt{c}_n\ \mathtt{ss}\ \mathtt{zz}) \to^2 \mathtt{fst}(\overbrace{\mathtt{ss}(\ldots\ (\mathtt{ss}}^{n}\ \mathtt{zz})\ldots)$$

いま，各 j に対して，$\mathtt{ss}(\mathtt{pair}\ \mathtt{c}_{j-1}\ \mathtt{c}_j)$ の簡約列は次のように書けるので，定数オーダーで簡約できることが分かる:

$$\mathtt{ss}(\mathtt{pair}\ \mathtt{c}_{j-1}\ \mathtt{c}_j) \to \mathtt{pair}(\mathtt{snd}(\mathtt{pair}\ \mathtt{c}_{j-1}\ \mathtt{c}_j))(\mathtt{plus}\ \mathtt{c}_1\ \mathtt{snd}(\mathtt{pair}\ \mathtt{c}_{j-1}\ \mathtt{c}_j))$$
$$\to^6 \mathtt{pair}(\mathtt{snd}(\mathtt{pair}\ \mathtt{c}_{j-1}\ \mathtt{c}_j))(\mathtt{plus}\ \mathtt{c}_1\ \mathtt{c}_j)$$
$$\to^5 \mathtt{pair}(\mathtt{snd}(\mathtt{pair}\ \mathtt{c}_{j-1}\ \mathtt{c}_j))(\mathtt{c}_{j+1}) \to^6 \mathtt{pair}\ \mathtt{c}_j\ \mathtt{c}_{j+1}$$

以上より $(\overbrace{\mathtt{ss}(\ldots\ (\mathtt{ss}}^{n}\ \mathtt{zz})\ldots)$ は $\mathcal{O}(n)$ のオーダーで $\mathtt{pair}\ \mathtt{c}_{n-1}\ \mathtt{c}_n$ に簡約できる．二つ組の射影は定数オーダーで評価できるので，題意が示される．□

λ 計算でのプログラミング: リスト (演習 5.2.8)

λ 計算で cons-list を表現することを考える.

演習 5.2.8

λ 計算において，リストはそれ自身の fold 関数である．たとえば，リスト [x, y, z] は λc.λn.c x (c y (c n)) なる関数として書ける．nil, cons, isnil, head, tail 関数を書け.

まず nil は定数なので，nil = λc.λn.n.

cons は要素 h とリスト t を取り h :: t を返す:
cons = λh.λt.λc.λn.c h(t c n).

isnil をリストに適用したとき，リストに n のみが含まれていれば tru，c および値が一つ以上含まれていれば fls を返す．よって
isnil = λt.t (λx.λb.fls) tru.

head 関数は，リストのコンストラクタ c を一斉に真理値 tru に，定数 n を fls に置き換えることで実現できる[1]．よって head = λt.t tru fls.

[1]本来は partial である.

λ 計算でのプログラミング: リスト (演習 5.2.8)

最後に tail を求めよう．prd 同様にタプリングを使って書けるのだから，

- nil に対しては pair nil nil を
- cons h t に対しては pair t (cons h t) を

返すように作ればよい．したがって，次のように定義できるとわかる:

tail 関数

tail $=\lambda$t.fst(t cc nn)．ただし

- nn $=$ pair nil nil
- cc $=\lambda$h.λp.pair(snd p)(cons h (snd p))

λ 計算でのプログラミング: 計算体系の拡張

純粋な λ 計算の体系 λ を，型無し算術式 **NB** で定義した式や値をプリミティブとして組み込んで拡張したものを **λNB** と呼ぶ．組み込んだプリミティブと Church ブール値/数を相互変換する関数は次のように定義できる:

λNB でのプリミティブとその表現間の相互変換

- Chuch ブール値からプリミティブ:
  $\texttt{realbool} = \lambda\texttt{b.b true false}$
- プリミティブから Church ブール値:
  $\texttt{churchbool} = \lambda\texttt{b.if b then tru else fls}$
- Church 数からプリミティブ: $\texttt{realnat} = \lambda\texttt{m.m}(\lambda\texttt{x.succ x})\ \texttt{0}$
- プリミティブから Church 数;

$$\begin{aligned}\texttt{churchnat} = &\lambda\texttt{m.if }(\texttt{iszero m})\texttt{ then } \texttt{c}_0\\ &\texttt{else scc}(\texttt{churchnat }(\texttt{pred m}))\end{aligned}$$

ただし最後の定義において，のちに導入する不動点コンビネータを暗に用いた．

プリミティブを導入することで，Church ブール値/数に関数を適用したものと，その結果そのものが等しいことが容易に分かる．

次の例を考える: $\mathtt{scc}\ \mathtt{c}_1$．この項は $\mathtt{c}_2$ に評価されない．なぜなら，

$$\mathtt{scc}\ \mathtt{c}_1 = (\lambda \mathtt{m}.\lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{s}(\mathtt{m}\ \mathtt{s}\ \mathtt{z}))\mathtt{c}_1 \to \lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{s}((\lambda \mathtt{s}'.\lambda \mathtt{z}'.\mathtt{s}'\ \mathtt{z}')\mathtt{s}\ \mathtt{z}) \nrightarrow$$

となるから．値呼びで項が簡約できるのは項が $(\lambda \mathtt{x}.\mathtt{s}_{12})\mathtt{s}_2$($\mathtt{s}_2$ は値) の形であることなので，ここで行き詰まる．

もちろん，任意の項 $\mathtt{v}, \mathtt{w}$ について $(\mathtt{scc}\ \mathtt{c}_1)\mathtt{v}\ \mathtt{w} = \mathtt{c}_2\ \mathtt{v}\ \mathtt{w}$[2]が成りたつ，つまりこれら 2 つの項は振る舞い等価 である．さらに $\mathtt{equal}\ (\mathtt{scc}\ \mathtt{c}_1)\ \mathtt{c}_2$ も成り立つ．

同様のことは，煩雑な過程を経てたとえば $\mathtt{times}\ \mathtt{c}_2\ \mathtt{c}_2$ と $\mathtt{c}_4$ に対してもいえる．しかし，前者を realnat によりプリミティブな数に変換する方がより直接的である．

[2]つまり η 同値である．

λ 計算でのプログラミング: 再帰

- λ 計算においては，正規形へ評価できない項が存在する．
  - たとえば，発散コンビネータ omega $= (\lambda x.x\ x)(\lambda x.x\ x)$ について，omega $\to$ omega．
  - 正規形に評価できない項は，発散するという．
- omega を一般化し，関数の不動点を表わす以下のコンビネータが定義できる．

不動点コンビネータ

不動点コンビネータ fix を，$\mathtt{fix} = \lambda f.(\lambda x.f(\lambda y.x\ x\ y))(\lambda x.f(\lambda y.x\ x\ y))$ と定める．

λ 計算でのプログラミング: Church 数のリスト

演習 5.2.11

fix とリストの表現，すなわち任意の $x_1, \ldots, x_n$ に対し $[x_1, x_2, \ldots x_n] = \lambda c.\lambda n.c\ x_1(c\ x_2(\ldots(c\ x_n\ n)\ldots))$ とを用いて，Church 数のリストの総和を計算する関数を書け．

たとえば Haskell では，整数リストの総和を計算する関数 $\mathtt{sum} :: [\mathtt{Int}] \to \mathtt{Int}$ は次のように書けた:

$$\mathtt{sum}\ [] = 0$$

$$\mathtt{sum}\ (x : xs) = x + \mathtt{sum}\ xs$$

これをそのまま fix コンビネータで書き下せばよい．演習 5.2.8 の isnil, head, tail 関数を使う．

$$\mathtt{h} = \lambda f.\lambda t.\mathtt{test}\ (\mathtt{isnil}\ t)\ (\lambda x.c_0)\ (\lambda x.(\mathtt{plus}\ (\mathtt{head}\ t)\ (f(\mathtt{tail}\ t))))\ c_0$$

$$\mathtt{sum} = \mathtt{fix}\ f$$

λ 計算でのプログラミング: 表現

Church 数が” 普通の数” を表わすとはどういうことだろうか.

- 数を，型無し算術式において定めた定数 `0`，数の上の演算 `succ`, `pred`, `iszero` からなるものとして特徴付ける.
- すると，今までの観察から，Church 数上の演算 `scc`, `prd`, `iszro` は Church 数上における `succ`, `pred`, `iszero` と同様に振る舞う.

ひいては，数をブール値に写すプログラムに対し，数を Church 数に，ブール値を Church ブール値に置き換えてプログラムを評価したとしても同じ結果を得る.

(定義) 項

$\mathcal{V}$ を変数名の可算集合とする．項の集合は以下を満たす最小の集合 $\mathcal{T}$:

1. 任意の $x \in \mathcal{V}$ に対して $x \in \mathcal{V}$.
2. $t_1 \in \mathcal{T}, x \in \mathcal{V}$ ならば $\lambda x.t_1 \in \mathcal{T}$.
3. $t_1, t_2 \in \mathcal{V}$ ならば $t_1\ t_2 \in \mathcal{V}$.

(定義) 自由変数

項 t の自由変数の集合 $\mathrm{FV}(t)$ を，以下のように定義する:

1. $\mathrm{FV}(x) = \{x\}$
2. $\mathrm{FV}(\lambda x.t_1) = \mathrm{FV}(t_1) \setminus \{x\}$
3. $\mathrm{FV}(t_1\ t_2) = \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2)$.

以下の定義では，項は束縛変数名を除いて同一視する，つまり $\mathcal{T}$ は α 同値類で割られているものとする．

代入

項 s の項 t の変数 x への代入を $[x \mapsto s]\, t$ とかき，次のように定義する．

$$[x \mapsto s]\, x = s$$

$$[x \mapsto s]\, y = y \qquad \text{if } x \neq y$$

$$[x \mapsto s]\, (\lambda y.t_1) = \lambda y.\, [x \mapsto s]\, t_1 \qquad \text{if } y \neq x \text{ and } y \notin \mathrm{FV}(s)$$

$$[x \mapsto s]\, (t_1\ t_2) = ([x \mapsto s]\, t_1)([x \mapsto s]\, t_2)$$

特に抽象への代入で条件を緩めたら何が起こるか観察してみよう．

形式的議論: 代入

1. 一切条件を課さないとき．α 同値な項に対する置換の結果が異なってしまう．
   - たとえば，$[\mathsf{x} \mapsto \mathsf{x}]\,(\lambda \mathsf{x}.\mathsf{x}) \neq [\mathsf{x} \mapsto \mathsf{y}]\,(\lambda \mathsf{y}.\mathsf{y})$
2. $\mathsf{y} \notin \mathrm{FV}(\mathsf{s})$ を課さないとき．
   - たとえば，$[\mathsf{x} \mapsto \mathsf{z}]\,(\lambda \mathsf{z}.\mathsf{x}) = \lambda \mathsf{z}.\mathsf{z}$ のような変換が許されてしまう．つまり，元々自由だったはずの変数が代入後に束縛されてしまう．

単純にこの定義に基づくと，たとえば $[\mathsf{x} \mapsto \mathsf{y}\ \mathsf{z}]\,(\lambda \mathsf{y}.\mathsf{x}\ \mathsf{y})$ は代入が行えないことになる．そこで，以下束縛変数名で項は同一視することにする．そうすれば，$[\mathsf{x} \mapsto \mathsf{y}\ \mathsf{z}]\,(\lambda \mathsf{y}.\mathsf{x}\ \mathsf{y}) \equiv [\mathsf{x} \mapsto \mathsf{y}\ \mathsf{z}]\,(\lambda \mathsf{w}.\mathsf{x}\ \mathsf{w}) \to \lambda \mathsf{w}.\mathsf{y}\ \mathsf{z}\ \mathsf{w}$ となり，代入操作は全域的になる．

値呼びの操作的意味論

まず，値呼びにおける構文要素を定める．

値呼びにおける構文要素

項は以下の BNF で定まる:

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

値は λ 抽象値である．したがって，以下の BNF で定まる:

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

次に評価関係を定めよう．値呼びで簡約が許されるのは最も外側の redex である．そうでない項は，はじめ左側，次いで右側の部分項が値となるまで簡約されるまで簡約される．定式化すると次のようになる．

値呼びにおける評価関係

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}\ (\text{E-APP1})$$

$$\frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'}\ (\text{E-APP2})$$

$$\frac{}{(\lambda x.t_{12})\ v_2 \to [x \mapsto v_2]\ t_{12}}\ (\text{E-APPABS})$$

E-APP1 により，関数適用は関数部分が値となるまで評価されたのち，E-APP2 によって左辺が値となるまで評価が進む．最後に E-APPABS により右辺が値となった redex のみ評価される．

λNB への行き詰まり項の導入

演習 3.5.7

演習 3.5.16 で導入した操作的意味論では行き詰まり状態がエラー項に評価されるようにした．これを λNB に拡張せよ．

ブール式の操作的意味論

ブール値でない正規形の構文要素 badbool を

$$\mathtt{badbool} ::= \mathtt{nv} \mid \mathtt{wrong}$$

と定める．操作的意味論は

$$\frac{}{\mathtt{if\ true\ then\ } t_1 \mathtt{\ else\ } t_2 \to t_1}\ \text{E-IFTRUE}$$

$$\frac{}{\mathtt{if\ false\ then\ } t_1 \mathtt{\ else\ } t_2 \to t_2}\ \text{E-IFFALSE}$$

$$\frac{t_1 \to t_1'}{\mathtt{if\ } t_1 \mathtt{\ then\ } t_2 \mathtt{\ else\ } t_3 \to \mathtt{if\ } t_1' \mathtt{\ then\ } t_2 \mathtt{\ else\ } t_3}\ \text{E-IFTRUE}$$

$$\frac{}{\mathtt{if\ badbool\ then\ } t_1 \mathtt{\ else\ } t_2 \to \mathtt{wrong}}\ \text{E-IF-WRONG}$$

数式の操作的意味論

構文要素 badnat を $\mathtt{badbool} ::= \mathtt{true}|\mathtt{false}|\mathtt{wrong}$ と定める．操作的意味論は

$$\frac{t_1 \to t_1'}{\mathtt{succ}\ t_1 \to \mathtt{succ}\ t_1'}\ \text{E-SUCC}$$

$$\frac{t_1 \to t_1'}{\mathtt{pred}\ t_1 \to \mathtt{pred}\ t_1'}\ \text{E-PRED}$$

$$\frac{}{\mathtt{pred}\ 0 \to 0}\ \text{E-PREDZERO}$$

$$\frac{}{\mathtt{pred}(\mathtt{succ}\ nv_1) \to nv_1}$$

$$\frac{t_1 \to t_1'}{\mathtt{pred}\ t_1 \to \mathtt{pred}\ t_1'}\ \text{E-PRED}$$

$$\mathtt{iszero}\ 0 \to \mathtt{true}$$

$$\mathtt{iszero}(\mathtt{succ}\ nv_1) \to \mathtt{false}$$

$$\mathtt{iszero}\ t_1 \to \mathtt{iszero}\ t_1'$$

$$\mathtt{pred}\ \mathtt{badnat} \to \mathtt{wrong}$$

$$\mathtt{succ}\ \mathtt{badnat} \to \mathtt{wrong}$$

$$\mathtt{iszero}\ \mathtt{badnat} \to \mathtt{wrong}$$

演習 5.3.7

まず，値呼びにおける行き詰まり項は，(1) 抽象でない，(2) 値への値でない項の適用，(3) 値でない項への項の適用の 3 通り．したがって，これらを含む構文要素 badlam を定義すればよい:

$$\mathsf{badlam} ::= \mathsf{na} | \mathsf{na\ t} | \mathsf{v\ na} | \mathsf{wrong}$$
$$\mathsf{na} ::= \mathsf{x} | \mathsf{t}_1\ \mathsf{t}_2$$

いま考えているのは **NB** を含むので，

$$\mathsf{if\ badlam\ then\ t}_1\ \mathsf{else\ t}_2 \to \mathsf{wrong} \qquad \mathsf{succ\ badlam} \to \mathsf{wrong}$$
$$\mathsf{pred\ badlam} \to \mathsf{wrong} \qquad \mathsf{iszero\ badlam} \to \mathsf{wrong}$$
$$\mathsf{badlam} \to \mathsf{wrong} \qquad \mathsf{churchbool\ badbool} \to \mathsf{wrong}$$
$$\mathsf{churchnat\ badnat} \to \mathsf{wrong} \qquad \mathsf{realbool\ badlam} \to \mathsf{wrong}$$
$$\mathsf{realnat\ badlam} \to \mathsf{wrong}$$