Passion が大事

mmichish

1 Intro

プログラミング言語の観点から見たとき、今までやってきた simply-typed λ-calculus とその拡張のお話だけではプログラミング言語としては成り立たなくて、評価 (evaluation) の notation とかが必要になるわね。今まで equational theory なんかを確認したけど、評価はこれらから直ちに導かれるものでもないし、評価のアプローチ自体にはいくつか考えられるからちょっと見ていくわ。そんな中でこの章では再帰定義の導入とその semantics を考えていきたいぞ。プログラミング言語にはほしいもんね。semantics をいくつか定義することができるので、それぞれがどんな感じに関係しているのかを調べていくよ。さて最初に、反復を表現する構築子や再帰定義の話に加えて、それ以外にもベーシックな型 (integer, boolean) とか定数があると便利だよね。というお話から始めて、どういう感じに再帰定義していくといいかという話をしていくぞ。

はじめにいくつか再帰定義の例を見ていく。例えば階乗関数の再帰定義の例で、ML の場合だとこんな感じに書く。

$$val\ rec\ f\ =\ fn\ n : int\ =>\ if\ n\ =\ 0\ then\ 1\ else\ n\ *\ f\ (n\ -\ 1)$$

この宣言だと、f という識別子に対して、等式の右辺を定義している。λ の代わりに'fn' を利用し binding variable は'n:int' で、'.' でなく'⇒' で表現するイメージ。つまり f は int 型の値を取って、'if n = 0 ...' を計算する関数として定義され、そのなかで f を (n-1) を伴って呼び出している。
Lisp（正確には Scheme）だとこう

$$\begin{array}{l}(define\ f\ (lambda\ (n)\\ \qquad (if\ (=\ n\ 0)\\ \qquad\qquad 1\\ \qquad\qquad\ (*\ n\ (f\ (-\ n\ 1))))))\end{array}$$

abstraction における型タグが存在しないが、まぁそれ以外は比較そのまま理解できるだろう。これを見ると、現状ではまだちゃんと if とかを表現する定数みたいなのは出てきてないが、気持ちとしては simply-typed λ calculus with products and constants で定義できそうな雰囲気になる（らしいです、僕はなりません）

$$\lambda n : C_{int}.\ c_{if}(c_{=}(n, c_0),\ c_1,\ c_{*}(n, f(c_{-}(n, c_1)))) \tag{1}$$

これは気持ちとしては次のような感じ。

- $\lambda n : C_{int}$: 渡される値 n で型は C_{int}. Constant Type C の一つ
- $c_{if}\ c_0\ c_1\ c_{*}\ c_{-}$: これらはそれぞれ term constant. 気持ちとしては if とか二項演算 $*$ とかの表現で、これらの挙動が評価のところの話に関わってきそうな気持ち
- この式では、term としては、たくさんの products で成立している感じで、例えば Appl + Prod で構成される、$c_{=}(n, c_0)$ は, 値 n と c_0（定数の 0）を二項演算=で比較している。

上記気持ちにより、まぁ確かにそれらしい syntax を決めて、それに対して評価方法が定義されていれば雰囲気計算できそうな気持ちはある。passion が大事。これはちょっとよくわからなかったが、なんか constants が'curried'type であるとか適当なことを言うと、なんか知らないが product を使わずに以下のようにかけるらしい。ちょっとわからん（すまん）。 → snumajir が教えてくれた。curry 化のことらしい。なるほど。

$$\lambda n : C_{int}.\ c_{if}(c_{=}(n)(c_0))(c1)(c_{*}(n)(f(c_{-}(n)(c_1))))$$

こう見ると、条件演算子の if、二項演算子の =, *, - が与えられていると、これは下記のような式になりそうな気になる。

$$\begin{array}{l}(define\ f\ (lambda\ (n)\\ \quad (((if\ ((=\ n)\ 0))\\ \quad\quad 1)\\ \quad\quad\quad ((*\ n)\ (f\ ((-\ n)\ 1))))))\end{array}$$

ただこれ、まぁ中身はいいとして、f の定義ってどう行われてるか不明瞭だよね、って問題がある。例えば、先に見た ML の例とかだと、"val rec"とかいう notation を使って f の定義を与えていたわけだけど、現状我々の見てきた λ-calculus の中で同じようなことしようとしたときに使えるのは Abstraction の binding しかない。ちょっと Abstraction ではつらみがあるので、再帰定義のためにいくつかの新しい定数を導入したいね。途端にガッツリ変わるが、例えば下記のような感じ。

$$\boldsymbol{Y}\lambda f : \boldsymbol{num} \to \boldsymbol{num}.\lambda n : \boldsymbol{num}.\ \boldsymbol{if}\ n\ =\ 0\ \boldsymbol{then}\ 1\ \boldsymbol{else}\ n * f(n-1) \tag{2}$$

ここで、$\boldsymbol{Y}$ はこのような型の定数

$$((\boldsymbol{num} \to \boldsymbol{num}) \to (\boldsymbol{num} \to \boldsymbol{num})) \to (\boldsymbol{num} \to \boldsymbol{num})$$

新しい binding 演算子を表現するために高階関数（$\boldsymbol{Y}$）と定数たちを利用して、追加の syntax を導入しないように避ける感じ。（定数も高階関数もこれまでの枠組みの中でなんとかなるからね。）

実際に bind することを考えると、term を名前にバインドする構文は、λ では Application で表現する。例えば、M が階乗関数で、プログラム N の中の関数名 f に M を bind したい場合、application を利用して

$$(\lambda f.N)M$$

と書くよね。ただいくつかの言語ではこれらは別の構文として用意されていて、

$$N\ where\ f\ =\ M$$

とか

$$let\ f\ =\ M\ in\ N$$

とか書く。ただし、この場合はちょっと注意が必要で、例えば M が 2 の例だと次のようになり、

$$let\ f\ =\ \boldsymbol{Y}\lambda f : \boldsymbol{num} \to \boldsymbol{num}.\lambda\ n : \boldsymbol{num}.\ \boldsymbol{if}\ n\ =\ 0\ \boldsymbol{then}\ 1\ \boldsymbol{else}\ n * f(n-1)\ in\ N$$

これは M の中に f を束縛する λ f が存在するため、'let f' が f を束縛しないことが明白だけど、M が 1 のときは次のようになり、

$$let\ f\ =\ \lambda n : C_{int}.\ c_{if}(c_{=}(n, c_0),\ c_1,\ c_{*}(n, f(c_{-}(n, c_1))))\ in\ N$$

この場合は、M の中の f の出現が'let f' に束縛されているかどうか明確ではない感じになっている。関数 f が再帰定義のつもりなら、$\boldsymbol{M}$ の中の f の出現は、'let f' の f に束縛されてほしいし、逆に再帰定義でない場合は、f は'let f' に束縛されないでほしいため、明確に使い分けられないとまずい。なのでこれを明確にするために、「束縛するよ」という notation を新しく導入する。

$$letrec\ f\ =\ M\ in\ N \tag{3}$$

この場合、letrec に続く f は項 $\boldsymbol{M}$ の中の f の出現を bind する。結果的に、下記の notation と同じ意味になる。

$$let\ f\ =\ \boldsymbol{Y}\lambda f.\ M\ in\ N$$

多くのプログラミング言語ではこれを使ってる。

一方で、論理的な利用の面からするとちょっと'letrec' notation は式 3 でいうところの項 N を必ず含める必要がありしんどみがある。$\boldsymbol{Y}$ という notation を利用する場合は、N を無視して、再帰定義関数を単一の式を書くことができるから嬉しい。これを Constant で表現する方法よりも、もう新しい binding operator（μ）を導入したほうがシンプルらしいのでそうする。これは λ-binder（Abstraction の λ）に似てるけど、再帰定義を意味するよ。（実質 $\boldsymbol{Y}\lambda$ を μ で置き換えればいい。）

$$\mu f : num \to num.\lambda n : num.\ if\ n\ =\ 0\ then\ 1\ else\ n * f(n-1) \tag{4}$$

そもそも再帰定義を Application の特殊ケースとして考えるのではなく、新しいケースとして捉えてしまったほうが楽。と、良く μ-notation を支持するときに言われるけど、まぁそれ以外もこっから先の議論のために十分役立つと思うよ。

ここまで、いくつかの再帰定義の Syntax をいくつか見てきたけど、semantics についてはやってないよね。まぁこれの評価方法って明示的に言わずもがなでよく利用されてて、例えば階乗関数の場合はこう

$$f(n) = \begin{cases} 1 & if\ n = 0 \\ n * f(n-1) & if\ n > 0. \end{cases}$$

これは例えば、

$$\begin{aligned} f(3) &= 3 * f(2) \\ &= 3 * 2 * f(1) \\ &= 3 * 2 * 1 * f(0) \\ &= 3 * 2 * 1 * 1 \\ &= 6 \end{aligned}$$

こう計算できるわけだけど、これをどう formal に説明できるかが本章の目的である。議論の前に簡単に言語を定義して、それに基づいて話を進めていくよ。

1.1 A Programming Language for Computable Functions

次に示すのは、simple higher-order Programming language for Computable Functions (PCF) という、型付きラムダ計算の拡張だぞ。まずは syntax の定義はこんな感じ。

$$
\begin{aligned}
t &::= num \mid bool \mid t \to t \\
M &::= 0 \mid true \mid false \mid \\
&\quad succ(M) \mid pred(M) \mid zero?(M) \mid if\ M\ then\ M\ else\ M \mid \\
&\quad x \mid \lambda x : t.M \mid MM \mid \mu x : t.M
\end{aligned}
$$

ざっくり箇条書きにすると下記のような感じらしいぞ

- simply-typed λ-calclus を含んでいる
- ground type は 2 つ。num と $bool$ のみ。
- PCF syntax の扱いは基本的に Section2.1 で取り扱った λ-calculus になる。
  - 変数や項の naming や、項の parsing-convention は λ-calculus と同じ。
  - PCF term とか呼ぶけど、文脈から明白なときは term と呼ぶ。
- 新しい項
  - $successor$, $predecessor$, $test\ for\ zero$
  - mix-fix operator (condition / $if\ then\ else$ の項)
    - (ex.) $if\ L\ then\ M\ else\ N$
    - L : condition / test
    - M : first branch
    - N : second branch
    - L, M に関する parse は問題ないだろうが、N は可能な限りスコープを広く取る。
    - (ex.) $if\ x\ then\ f\ else\ gy$ の場合、$(if\ x\ then\ f\ else\ g)(y)$ ではなく、$if\ x\ then\ f\ else\ (g(y))$ として解釈する
- 新しい binding operator の μ、参戦。
  - α-equivalence と parsing-convention の定義は λ と同じ感じ。
  - $\mu x : t.M$ という式ががあったとき、x が binding occurence, M が body
  - μ の後の x の出現は、項 M の中の自由な x 全ての出現に対する binding occurence である。

構文木の context substitution は以下のような感じになる。

- $\{M/x\}0$ is 0. $\{M/x\}true$ is $true$. $\{M/x\}false$ is $false$
- $\{M/x\}zero?(N)$ is $zero?(\{M/x\}N)$. $\{M/x\}succ(N)$ is $succ(\{M/x\}N)$. $\{M/x\}pred(N)$ is $pred(\{M/x\}N)$.
- $\{P/x\}if\ L\ then\ M\ else\ N$ is $if\ \{P/x\}L\ then\ \{P/x\}M\ else\ \{P/x\}N$.
- N が Abstraction で、$\mu x : t.L$ (bound variable $= x$) のとき、$\{M/x\}N$ is N.
- N が Abstraction で、$\mu y : t.L$ (bound variable $= y \neq x$) のとき、$\{M/x\}N$ is $\mu y : t.\{M/x\}L$

PCF の型付け規則（typing rule）は simply-typed λ-calculus + α になる。

Typing Rules for PCF (include Table 2.1)

Proj	$H, x : t, H' \vdash x : t$
Abs	$\dfrac{H, x : s \vdash M : t}{H \vdash \lambda x : s.M : s \to t}$
Appl	$\dfrac{H \vdash M : s \to t \quad H \vdash N : s}{H \vdash M(N) : t}$
Zero	$H \vdash 0 : num$
True	$H \vdash true : bool$
False	$H \vdash false : bool$
Pred	$\dfrac{H \vdash M : num}{H \vdash pred(M) : num}$
Succ	$\dfrac{H \vdash M : num}{H \vdash succ(M) : num}$
IsZero	$\dfrac{H \vdash M : num}{H \vdash zero?(M) : bool}$
Cond	$\dfrac{H \vdash L : bool \quad H \vdash M : t \quad H \vdash N : t}{H \vdash if\ L\ then\ M\ else\ N\ :\ t}$
Rec	$\dfrac{H, x : t \vdash M : t}{H \vdash \mu x : t.M : t}$

- **0** は *num* 型を持ち、***true***、***false*** は *bool* 型を持つ
- *num* 型の項（式）M が与えられ得たとき、$pred(M)$ と $succ(M)$ は *num* 型で、$zero?(M)$ は *boolean* 型である。
- *Cond*(*conditional expression*) では、*condition* L が *bool* 型、*branch* M、N が同じ型 t を持つ。Condition 自身もこれと同じ型 t を持つ。
- type assignment に $x : t$ がある状態で t に片付けされる M に対して、$\mu x : t.M$ は同じ型 t を持つ。

Example

PCF を利用する例として、下記のような式を考えることができる。

$$Minus \equiv \mu\ minus :\ num \to num \to num.\ \lambda x : num.\ \lambda y : num.$$
$$if\ zero?(y)\ then\ x\ else\ minus(pred(x))(pred(y))$$

複雑な関数を考えるときは、*local definition* を利用するのが有効だぞ。例えば、乗算の定義をする場合は、*local definition* として加算を定義しておくと楽だぞ。（なんか local definition の仕方も色々ありそうな気がしているが、本書では下記のように書いていた。）

$$
\begin{aligned}
&(\lambda plus :\ num \to num \to num.\\
&\quad \mu times :\ num \to num \to num.\ \lambda x : num.\ \lambda y : num.\\
&\qquad if\ zero?(y)\ then\ 0\\
&\qquad else\ plus(x)(times(x)(pred(y)))\\
&)\ \mu plus :\ num \to num \to num.\ \lambda.x : num.\ \lambda y : num.\\
&\quad if\ zero?(y)\ then\ x\\
&\quad else\ plus(succ(x))(pred(y))
\end{aligned}
$$

実際にこんな感じの式の計算例は、次の equational rule の定義を与えた後に行うことができる。

4.1 *Lemma*

1 If $H \vdash M : s$ and $H \vdash M : t$, then $s \equiv t$

2 If $H, x : s \vdash M : t$ and $H \vdash N : s$, then $H \vdash [N/x]M : t$

これらは別ノートにまとめてあるのでそちらを参照。

次に、PCF の *equational theory* を考える。*equational theory* は λ-calculus でやった Table2.2 に加えて、Table4.2 のものが追加される。まず復習として、以前やった Equational Rule の復習。

***Equational Rule* (*Table* 2.2)**

Axiom
$$\frac{(H \triangleright M = N : t) \in T}{T \vdash (H \triangleright M = N : t)}$$

Add
$$\frac{T \vdash (H \triangleright M = N : t) \quad x \notin H}{T \vdash (H, x : s \triangleright M = N : t)}$$

Drop
$$\frac{T \vdash (H, x : s \triangleright M = N : t) \quad x \notin Fv(M) \cup F(N)}{T \vdash (H \triangleright M = N : t)}$$

Permute
$$\frac{T \vdash (H, x : s, y : s, H' \triangleright M = N : t)}{T \vdash (H, y : s, x : s, H' \triangleright M = N : t)}$$

Refl
$$\frac{H \vdash M : t}{T \vdash (H \triangleright M = M : t)}$$

Sym
$$\frac{T \vdash (H \triangleright M = N : t)}{T \vdash (H \triangleright N = M : t)}$$

Trans
$$\frac{T \vdash (H \triangleright L = M : t) \quad T \vdash (H \triangleright M = N : t)}{T \vdash (H \triangleright L = N : t)}$$

Cong
$$\frac{T \vdash (H \triangleright M = M' : s \to t) \quad T \vdash (H \triangleright N = N' : s)}{T \vdash (H \triangleright M(N) = M'(N') : t}$$

ξ
$$\frac{T \vdash (H, x : s \triangleright M = N : t)}{T \vdash (H \triangleright \lambda x : s.M = \lambda x : s.N : s \to t)}$$

β
$$\frac{H, x : s \vdash M : t \quad H \vdash N : s}{T \vdash (H \triangleright (\lambda x : s.M)(N) = [N/x]M : t)}$$

η
$$\frac{H \vdash M : s \to t \quad x \notin Fv(M)}{T \vdash (H \triangleright \lambda x : s.M(x) = M : s \to t}$$

次に、PCF の *equational theory* は次のように与える。

Equational Rule for Call by Name PCF

PredZero
$$\vdash (H \triangleright pred(0) = 0 : num)$$

PredSucc
$$\vdash (H \triangleright pred(succ(n)) = n : num)$$

ZeroIsZero
$$\vdash (H \triangleright zero?(0) = true : bool)$$

SuccIsNotZero
$$\vdash (H \triangleright zero?(succ(n)) = false : bool)$$

IfTrue
$$\frac{H \vdash M : t \quad H \vdash N : t}{\vdash (H \triangleright if\ true\ then\ M\ else\ N = M : t)}$$

IfFalse
$$\frac{H \vdash M; t \quad H \vdash N : t}{\vdash (H \triangleright if\ false\ then\ M\ else\ N\ = N : t)}$$

μ
$$\frac{H, x : s \vdash M : t}{\vdash (H \triangleright \mu x : t.M = [\mu x : t.M/x]M : t)}$$

Cong
$$\frac{\vdash (H \triangleright M = N : num)}{\vdash (H \triangleright pred(M) = pred(N) : num}$$

$$\frac{\vdash (H \triangleright M = N : num)}{\vdash (H \triangleright succ(M) = succ(N) : num)}$$

$$\frac{\vdash (H \triangleright M = N : num)}{\vdash (H \triangleright zero?(M) = zero?(N) : bool)}$$

$$\frac{\vdash (H, x : t \triangleright M = N : t)}{\vdash (H \triangleright \mu x : t.M = \mu x : t.N : t)}$$

$$\frac{\vdash (H \triangleright L = L' : boot) \quad \vdash (H \triangleright M = M' : t) \quad \vdash (H \triangleright N : t)}{\vdash (H \triangleright if\ L\ then\ M\ else\ N = if\ L'\ then\ M'\ else\ N' : t}$$

- 数値 n の *successor* の *predecessor* は n になる。0 の predecessor は 0 になる。
- 数値の *successor* に対する *zero?* は $false$、0 のとき $true$
- cond は、condition の項が $true$ ならば、condition 自体が first branch と等しくなり、false のときは second branch と等しくなる。
- recursion である、$\mu x : t$ は recursion それ自体を body の全ての x の出現に代入することで得られる *unwinding*(巻き戻し、再帰の意味かな?) と等しくなる。
- 上記のやつらは associated congluence rule を持つよ。

この *syntax*、*typing rule*, *equational rule* で $2 - 1 = 1$ を示すとこんな感じになるような気持ち。(minus は先に出てきたもの)。

$$
\begin{aligned}
minus(succ(succ(0)))(succ(0)) &= if\ zero?(succ(0))\ then\ succ(succ(0)) \\
&\quad else\ minus(pred(succ(succ(0))))(pred(succ(0))) \\
&= minus(pred(succ(succ(0))))(pred(succ(0))) \\
&= ifzero?(pred(succ(0)))\ then\ pred(succ(succ(0))) \\
&\quad else\ minus(pred(pred(succ(succ(0)))))(pred(pred(succ(0)))) \\
&= pred(succ(succ(0))) \\
&= succ(0)
\end{aligned}
\tag{5}
$$

ただ、*equational rule* はあくまである前提のもとに成り立つ *equality* を定めているだけであって、上記の例のような計算の手法については述べていないので、上記の通り現時点でこの式変形による 2-1=1 はお気持ち。

ということでここからはその計算の手法に当たる話をしていく。リダクションとか戦略とか、*Operational Semantics*, 操作的意味を与える、とかいう話。リダクションを定めると、ある式が与えられたときに定義したリダクションに基づいて評価することで、結果（それ以上リダクションされない形。*value*）を得ることができる！嬉しい !!

Table4.3 に今回決めた二項遷移関係（→）を示す。これが今回定義するリダクション規則になる。ちなみにこれは Application の rule を見るとわかるが、Call-by-Name の戦略になっている。（Application の右側（適用される側）を評価せず、そのまま substitution の処理に進んで評価が進んでいく。）

Transition Rules for Call by name Evaluation of PCF

$$\frac{M \to N}{pred(M) \to pred(N)} \qquad pred(0) \to 0$$

$$pred(succ(V)) \to V \qquad \frac{M \to N}{zero?(M) \to zero?(N)}$$

$$zero?(0) \to true \qquad zero?(succ(V)) \to false$$

$$\frac{M \to N}{succ(M) \to succ(N)} \qquad \frac{M \to N}{M(L) \to N(L)}$$

$$(\lambda x : t.M)(N) \to [N/x]M \qquad if\ true\ then\ M\ else\ N\ \to M$$

$$if\ false\ then\ M\ else\ N\ \to N \qquad \frac{L \to L'}{if\ L\ then\ M\ else\ N \to if\ L'\ then\ M\ else\ N}$$

$$\mu x : t.M \to [\mu x : t.M/x]M$$

$M \to^* V$ であるとき、項 M は値 V にのみ評価される。ここで、$\to^*$ は定義した $\to$ の transitive, reflexive closure である。また、value は以下の項のどれかである。

$$V\ ::=\ 0 \mid true \mid false \mid succ(V) \mid \lambda x : t.M \tag{6}$$

V の代わりに U とか W を使うよ。注意してほしいのは、全ての value はそれぞれに評価されるし、transition relation は決定的だということ。（上の定義に従って推移するという意味で決定的と言ってるのか？）

4.2, 4.3 *Lemma*

これらは別ノートにまとめてあるので、そちらを参照。Lemma4.2 では、リダクションが一意に決まることを言っているのでそれなりに大切そう。てかこれ成り立たつようにうまく定義するものだと思っている。Lemma4.3 は後で使う。

さて、リダクション規則が定まったので、先の例（minus）を例に取り上げて計算してみる。。。

すまん全部クソ真面目に計算するやつ手書きでやったけどこっちに書くのしんどかったので省略します。別ノートの冒頭に個人的な歴史的経緯から ***Add*** の例を書いたやつがあるので（***minus*** でなくてすまん）気になったらそっちを見てください。定義どおりにやるだけなので難しくないです。

ただ、実はこれって equality をリダクションをごっちゃにしているよね。リダクションは定義によって導出できるけど、これ自身はイコールでつないでいいみたいな議論はしていない。しちゃいなよ You、ということでこれ。

4.4 *Lemma*

これらは別ノートにまとめてあるので、そちらを参照。PCF の式のリダクションと *equality* との整合性を示す感じ。Theorem4.4 によってわかることは下記の 2 つで。

1 $H \vdash M : t$ と $H \to N$ は $H \vdash N : t$ を意味する（*Subject Reduction*）

2 equation に対する translation relation（$\to$）の健全性

Section1.2 の Simple Imperative Language で話したが、場合によっては relation $M \to^* V$ は新しい rule のセットによってもうちょい直接的に記述できる（*natural* (*operational*) *semantics* というらしい）。$M \Downarrow V$ と書いたとき、右側が term, 左側がその term を評価した結果の *value* である。これは Translation Relation みたいな 1 ステップの評価ルールというより、一気に評価していくイメージ（あってんのか？）。これから PCF でこれを考えていく。項のメタ変数としては、L, M, N を利用し、Value のメタ変数としては U, V, W を使うとのこと。

Natural Rules for Call − by − name Evaluation of PCF

$$0 \Downarrow 0 \qquad true \Downarrow true$$

$$false \Downarrow false$$

$$\frac{M \Downarrow 0}{pred(M) \Downarrow 0} \qquad \frac{M \Downarrow succ(V)}{pred(M) \Downarrow V}$$

$$\frac{M \Downarrow V}{succ(M) \Downarrow succ(V)} \qquad \frac{M \Downarrow 0}{zero?(M) \Downarrow true}$$

$$\frac{M \Downarrow succ(V)}{zero?(M) \Downarrow false} \qquad \lambda x : s.M \Downarrow \lambda x : s.M$$

$$\frac{M \Downarrow \lambda x : s.M' \quad [N/x]/M' \Downarrow V}{M(N) \Downarrow V} \qquad \frac{M_1 \Downarrow true \quad M_2 \Downarrow V}{if\ M_1\ then\ M_2\ else\ M_3 \Downarrow V}$$

$$\frac{M_1 \Downarrow false \quad M_3 \Downarrow V}{if\ M_1\ then\ M_2\ else\ M_3 \Downarrow V} \qquad \frac{[\mu x : t.M/x]M \Downarrow V}{\mu x : t.M \Downarrow V}$$

- Constants はそれ自身に評価する
- predecessor については項を先に評価する。その結果が zero なら predecessor も zero だし、n + 1 なら n
- successor についても項を先に評価する、その結果 n がなら successor は n + 1
- test for zero についても最初に項を評価し、結果が 0 なら値は true になり、successor なら値は false になる。
- abstraction はそれ自身に評価する。application は項 M に項 N を適用する場合、項 M から先に評価する。もしこのとき項 M を評価した結果が abstraction だった場合は項 N を代入した結果を評価する。
- condition の場合は test から評価する。その結果 true となったばあいは、condition の評価の値は first branch の評価の値になり、false の場合は second branch の評価の値になる。
- recursion の場合は、自分自身の body にある束縛変数に自分自身を代入して、その項を評価する。もしこの操作で結果があるとすれば、その評価の値＝ recursion の評価の値になる。

4.5 ***Lemma***
これは別ノートにまとめてあるのでそちらを参照。

TODO **:** すまんちょっとここはちゃんと理解できてないので解説してほしい。

次の証明 **4.6** ***Theorem*** に関わってくる話。この証明では、$\Downarrow$ と $\rightarrow^*$ が、PCF と value の間に同じ関係を定義しているぞ、ということを証明する。なんか、これは良い nature of argument の例って書いてあったけど、nature of argumentって何？固有名詞？俺の読解ミスかな？

さて、Theorem4.4 を振り返ってみると、これは 構造に関する帰納法 によって証明できた。この induction でうまくいくのは、Transition Rule for Call-by-Name PCF の定義の Rule の仮定が、その結論の subterm になっているためである。このような性質を持つ Transition Semantics を *Structual Operational Semantics* とも言うらしい。

さて問題はこっからで、次の証明を行う中で出てくるが、通常 Natural Semantics のルールを含むある事実の証明を行う場合、Structual Induction のみで証明することはできない（らしい。ここがちゃんとわかっていない）。曰く、仮定がその結論の Subterm になっていないケース（PCF の場合は μ-rule, まぁ確かに subterm にはなっていない）があるためであるとか。

そのため、構造に関する帰納法と、評価の長さに関する帰納法を同時に回すことでうまくやるよ、と書いてあったがなんでこれでやるといいのかもちゃんとわかっていない。マジですまない。教えて下さい。

4.6 *Theorem*

これは別ノートにまとめてあるのでそちらを参照。

4.7 *Corollary*

これは別ノートにまとめてあるのでそちらを参照。

Exercises
これは別ノートにまとめてあるのでそちらを参照。