

工学部専門科目「計算と論理」配布資料

単純型付ラムダ計算

五十嵐 淳

京都大学 大学院情報学研究科 通信情報システム専攻

cal15@fos.kuis.kyoto-u.ac.jp

<http://www.fos.kuis.kyoto-u.ac.jp/~igarashi/class/cal/>

October 20, 2015

Coq で, `Compute` によるプログラムの実行がどのようになされるか, また `reflexivity` を使って証明ができる条件 (すなわち両辺が等しいとはどういうことか) をより正確に理解するために, Coq の (特にプログラミング言語部分の) ベースとなっている理論である型付ラムダ計算 (typed λ -calculus) を導入する.

“calculus” もしくは computational calculus とは, (解析学ではなく) 「計算体系」という, 計算プロセスを表す理論的枠組みのことで, プログラムを一般化した項 (term) の変形を計算プロセスと見做そう, という考えに基いている.

1 自然数の加算と乗算

ここではまず導入として `S`, `0` で表現される自然数に対する加算・乗算を計算体系として表現することを考える.

1.1 項の構文

$$\begin{array}{l} \text{(terms) } M, N \in \text{Terms} ::= 0 \\ \quad \quad \quad \quad \quad \quad \quad | \text{ S } M \\ \quad \quad \quad \quad \quad \quad \quad | \text{ } M+N \\ \quad \quad \quad \quad \quad \quad \quad | \text{ } M*N \end{array}$$

ゼロ, サクセサ, 足し算とかけ算の二項演算を項として考える. 括弧は正確には構文要素ではないが, 結合を表すために適宜使用する. 括弧のない場合, 乗算は加算より強く結合¹し, どちらも左結合,²である. また, `S S 0` は `S (S 0)` としか読みようがないのだが, 後で行われる項の拡張を考えて, 括弧を書くことにする. また, `S M` は `+`, `*` よりも強く結合³する.

¹`M + N * P` は `M + (N * P)` を示す.

²`M + N + P` は, `(M + N) + P` を, `M * N * P` は, `(M * N) * P` を, 示す.

³`S M+N` は `(S M)+N` を示す.

1.2 簡約

簡約 (reduction) とは、計算の規則に従って項の単純化を行うことを指す。例えば、足し算の規則としては、 $0 + S\ 0$ が $S\ 0$ になる、といったことが考えられる。これを

$$0 + S\ 0 \longrightarrow S\ 0$$

と矢印を使って表す。数学的には、集合 *Terms* 上の二項関係 \longrightarrow を導入することになる。二項 M, N がこの関係で関係付けられていることを $M \longrightarrow N$ と書く。直感的な意味は「 M が 1 ステップで N に簡約される」ということである。

簡約関係は推論規則と呼ばれる形式を使って与えられる。以下は、「どんな項 M に対しても、 $0 + M \longrightarrow M$ という関係が成立する」ことを示す推論規則である。

$$0 + M \longrightarrow M \quad (\text{R-PLUSZ})$$

R-PLUSZ

は推論規則の名前である。この推論規則は M に適宜具体的なものをあてはめることで具体的な項の間の関係を導くための、関係の雛形・テンプレートとして機能する。例えば、 $M = S\ 0$ とすれば、

$$0 + S\ 0 \longrightarrow S\ 0$$

が導けるし、 $M = S\ (S\ (S\ 0))$ とすれば、

$$0 + S\ (S\ (S\ 0)) \longrightarrow S\ (S\ (S\ 0))$$

が導ける。

その他の足し算、かけ算に関する規則は以下の通りである。教科書の Coq での `plus`, `mult` の定義との類似を見てとってほしい。

$$(S\ M) + N \longrightarrow S\ (M + N) \quad (\text{R-PLUSS})$$

$$0 * M \longrightarrow 0 \quad (\text{R-MULTZ})$$

$$(S\ M) * N \longrightarrow N + M * N \quad (\text{R-MULTS})$$

これらの規則を使うと、

$$S\ 0 + S\ 0 \longrightarrow S\ (0 + S\ 0)$$

や

$$S\ 0 * S\ 0 \longrightarrow S\ 0 + 0 * S\ 0$$

といった関係が導ける。

このように加算・乗算の基本的計算ステップを推論規則として表現できるが、これらの右辺の次の計算ステップとして考えられる

$$S\ (0 + S\ 0) \longrightarrow S\ (S\ 0)$$

や

$$S\ 0 + 0 * S\ 0 \longrightarrow S\ 0 + 0$$

のように、式の一部 (部分式 (subexpression) と呼ぶ) に規則をあてはめて計算を進めるような関係を、上の規則だけでは導くことはできない。このような部分項の簡約を表現するために、例えば以下のような規則を導入する。

$$\frac{M \rightarrow M'}{S M \rightarrow S M'} \quad (\text{RC-Succ})$$

これは、これまでの規則と違い、水平線がひかれてその上下に $M \rightarrow N$ の形が書かれている。これは

上段の関係が言えたなら、下段の関係も導き出してよい

という意味で、上段は下段の関係を導き出すための前提条件となっている。例えば、

$$\begin{aligned} M &= 0 + S 0 \\ M' &= S 0 \end{aligned}$$

とすると、この規則は、

$$\begin{aligned} 0 + S 0 &\rightarrow S 0 \text{ が言えたなら} \\ S(0 + S 0) &\rightarrow S(S 0) \text{ をいってもよい} \end{aligned}$$

ということで、前提条件は規則 R-PLUSZ から満たされるので、結局 $S(0 + S 0) \rightarrow S(S 0)$ をいってもよいになる。このような、

1. 規則 R-PLUSZ より $0 + S 0 \rightarrow S 0$
2. 規則 RC-Succ より $S(0 + S 0) \rightarrow S(S 0)$

という、関係を確認するための推論過程を

$$\frac{\frac{0 + S 0 \rightarrow S 0}{S(0 + S 0) \rightarrow S(S 0)} \text{ R-PLUSZ}}{\text{RC-Succ}}$$

という推論規則をつなげた形で表現することがある。(一般には規則に前提条件が複数あって枝分かれする場合もあるので) このような表現を導出木と呼ぶ。

この規則 RC-Succ は、導出関係にある二項のまわりに $S()$ をつけても簡約関係にある、すなわち S の引数を簡約しても全体として簡約関係にあることを示している。同様な規則を項の種類だけ用意すると、結果として、

簡約できるところは、どこから計算してもよい

ということを表すことになる。部分項の簡約を許すための規則は以下のようなになる。

$$\frac{M \rightarrow M'}{S M \rightarrow S M'} \quad (\text{RC-Succ})$$

$$\frac{M \rightarrow M'}{M + N \rightarrow M' + N} \quad (\text{RC-PLUSL})$$

$$\frac{N \rightarrow N'}{M + N \rightarrow M + N'} \quad (\text{RC-PLUSR})$$

$$\frac{M \rightarrow M'}{M * N \rightarrow M' * N} \quad (\text{RC-MULTL})$$

$$\frac{N \rightarrow N'}{M * N \rightarrow M * N'} \quad (\text{RC-MULTR})$$

練習問題 1.1 以下の項 M_i について, $M_i \rightarrow N_i$ となる項 N_i を全て挙げよ. また, 関係の導出木を書け.

- $M_1 = S\ 0 + (S\ 0 * S\ 0)$
- $M_2 = (S\ 0 + S\ 0) * (S\ (S\ 0) + S\ 0)$
- $M_3 = S\ (S\ (S\ (S\ 0))) + 0$

練習問題 1.2 また, $S\ (S\ 0) * S\ 0$ が簡約されて $S\ (S\ 0)$ になる過程 (全て) を, $S\ (S\ 0) * S\ 0 \rightarrow M_1 \rightarrow M_2 \cdots \rightarrow S\ (S\ 0)$ となる M_i を列挙することで示せ.

2 関数とラムダ記法, ラムダ計算

プログラムにおいて, 似たような式・計算手順が複数の箇所で必要になった時には, 関数や手続きを定義して, 式・手順の再利用を図ることが一般的である. 数学でも

$$2^2\pi + 7^2\pi + 20^2\pi$$

と書く代わりに,

$$f(2) + f(7) + f(20) \quad \text{ただし } f(x) = x^2\pi$$

と書けば, 式の見通しがよくなる. ここで x はパラメータと呼ばれ, 使われる場所によって異なる部分を表す役割を担っている.

関数の概念は, (大学以降の?) 数学では集合の言葉を使って「入力と出力の対の集合」として捉えるが, 「入力から出力を計算する式」という見方も十分に直感的である. このような「計算可能な関数」を扱うための理論のひとつが λ 計算・ラムダ計算 (λ -calculus) であり, その中心となるのがラムダ記法と呼ばれる関数の記法である.

ラムダ記法は, $\lambda(\text{パラメータ}).(\text{式})$ という形で「パラメータを入力として式の計算結果を出力とする関数」を表現する. 上の例の f は $\lambda x.x^2\pi$ と書くことができる. このラムダ記法の特徴的な点は,

その関数自体に名前をつけずに関数を表現することができる

という点である. これにより, 関数の概念そのものと関数に名前をつけるという行為を切離すことができる.

ラムダ記法による関数に、その引数を与えた時 (関数を適用する, という) 関数の値は「パラメータを引数で具体化 (instantiate) すること」で得ることができる. すなわち, $f = \lambda x.x^2\pi$ とすると,

$$\begin{aligned} & f(2) + f(7) + f(20) \\ (\text{定義より}) &= (\lambda x.x^2\pi)(2) + f(7) + f(20) \\ (x \text{ を } 2 \text{ で具体化}) &= 2^2\pi + f(7) + f(20) \\ &\vdots \\ &= 453\pi \end{aligned}$$

という推論 (計算) が可能になる. ラムダ計算の体系は, ラムダ記法による関数, 関数適用によるパラメータ置換の仕組み (のみ) を形式的に実現したものであり, 特に, パラメータ置換 (これを代入 (*substitution*) と喚ぶ) こそが計算ステップである, という立場をとる.

ラムダ計算におけるこの計算ステップは β 簡約と呼ばれ, 以下のようなパターン (規則) で表される.

$$(\lambda x.M[x])N \longrightarrow M[N]$$

ここで, M, N はプログラムを表し, x は変数の名前を表す記号である. 表記 $M[x]$ や $M[N]$ は (0 個以上の) 「穴ボコ」が空いた項 M を考え, その穴ボコに x や N を入れたものを表している. これは要するに, x を仮引数・パラメータとする関数を実引数 N に適用すると, 結果として, 関数本体 M 中のパラメータ (の全ての出現) に N を代入したものになる, ということを表している.

3 型付ラムダ計算

型付ラムダ計算 (*typed λ -calculus*) は, ラムダ計算に Coq に見られるような型の概念を導入したものである. (本当は Coq が型付ラムダ計算に基いている, というべきだが.) 型付ラムダ計算は, ラムダ計算 (特に区別が必要な場合は「型無しラムダ計算」と呼ぶ) と同様, プログラムの理論研究での主要な道具であると同時に, 論理体系と計算体系の橋渡しをする「カリー・ハワードの同型対応」と呼ばれる見方を与える大事な体系である.

型付ラムダ計算には, 型としてどのようなものを考えるかによって様々なバリエーションがあるが, その中でも型として, nat , bool のような原始的な型 (primitive type, 基底型 (*base type*) というこも) と関数型 (のみ) を考える型付ラムダ計算を単純型付ラムダ計算 (*simply typed λ -calculus*)⁴ と呼ぶ. これは Coq でいうと, 大体 Basics.v と Induction.v の範囲で書いたプログラムに相当する.

ここでは単純型付ラムダ計算の定義を通じて, Coq におけるプログラムの (Compute による) 実行と (reflexivity で証明できる) 「等しさ」の概念を正確に把握する.

⁴ 「単純型付ラムダ計算」といった場合には, 厳密には, 再帰関数 (や, 場合によっては, nat , bool すらも!) を考えないことが多い. 本稿の計算体系は Plotkin の PCF と呼ばれる計算体系に近い.

3.1 型, ラムダ項の構文

まず, 単純型付ラムダ計算における型 (*type*) とラムダ項 (*λ -term*)⁵ を以下の構文で与える.

```

(types)     $S, T$  ::= nat
            |   bool
            |    $S \rightarrow T$ 

(variables)  $x, y$   ∈  {a, b, ...}
(terms)     $M, N$  ::=  $x$ 
            |   0
            |   S
            |   match  $M$  with 0 =>  $N_1$  | S  $x$  =>  $N_2$  end
            |   true
            |   false
            |   if  $M$  then  $N_1$  else  $N_2$ 
            |   fun  $x : T$  =>  $M$ 
            |   fix  $x(y : S) : T := M$ 
            |    $M_1 M_2$ 

```

- 型としては, 自然数の型 `nat`, 真偽値の型 `bool`, と関数型 $S \rightarrow T$ を考える. この S を引数型, T を返値型ということがある. \rightarrow は右結合する. 例えば, $T_1 \rightarrow T_2 \rightarrow T_3$ は $T_1 \rightarrow (T_2 \rightarrow T_3)$ のことであり, $(T_1 \rightarrow T_2) \rightarrow T_3$ ではない.
- 項としては, 変数, 自然数のコンストラクタと `match` による場合分け, 真偽値コンストラクタと `if` による場合分け, 関数 (`fun`), 再帰関数 (`fix`), 関数適用を考える. 関数の構文は Coq に合わせているが, 多くの文献では, $\lambda x : T. M$ と書かれる.
- 「関数の概念そのものと関数に名前をつけるという行為を切離すことができる」と書いたことに対応して,

```
Definition f (n:nat) (m:nat) : bool := ...
```

のような Coq の関数定義は

```
Definition f : nat -> nat -> bool :=
  fun (n:nat) => fun (m:nat) => ...
```

の略記である.

- 関数・再帰関数は括弧をその外側につけない限りできるだけ長く延ばして読む. また関数適用は左結合する. 例えば,

```
fun n : nat => plus n n
```

⁵ラムダ計算ではプログラムはラムダ項, もしくは単に項と呼ばれる.

は全体が関数項で

```
fun n : nat => (plus n) n
```

のことである.

- 再帰関数 $\text{fix } x(y : S) : T := M$ は Coq の Fixpoint で定義される関数に相当する. x が関数 (を再帰的に参照するため) の名前, y がパラメータ, S がパラメータの型, T が, 関数本体 M の型である.

```
Fixpoint plus (n : nat) (m : nat) : nat := ...
```

は

```
Definition plus : nat -> nat -> nat :=  
  (fix plus(n:nat) : nat -> nat := fun(m:nat) => ...)
```

と等価である.

Coq では, 適用した際に必ず停止するような関数しか書けないような制限が加わっているが, ここではその制限については扱わない. そのため,

```
fix f(x:nat) : nat := f x
```

のような, 一旦呼出されると計算が止まらない関数も (型がつく) 項として認められる.

3.2 簡約

計算過程を表す簡約関係は $M \longrightarrow N$ という形で書かれ, 「 M が 1 ステップで N に簡約される」と読む. 導入で紹介した β 簡約以外にも match や if による場合分けの処理が計算 1 ステップとして表される.

まず, β 簡約は, 先程と同様に

$$(\text{fun } x : T \Rightarrow M[x]) N \longrightarrow M[N] \quad (\text{R-BETA})$$

で表される. R-BETA を使うと, 例えば,

```
x = n  
T = nat  
M[x] = match x with 0 => S 0 | S n' => S (S x) end  
N = S 0
```

とすれば, 具体的な二項間の関係

```
(fun n : nat => match n with 0 => S 0 | S n' => S (S n) end) (S 0)  
  → match S 0 with 0 => S 0 | S n' => S (S (S 0)) end
```

が言える.

if や match の規則は β 簡約に比べれば幾分簡単である.

$$\text{match } 0 \text{ with } 0 \Rightarrow N_1 \mid S \ x \Rightarrow N_2 \text{ end} \longrightarrow N_1 \quad (\text{R-MATCHZ})$$

$$\text{match } S \ M \text{ with } 0 \Rightarrow N_1 \mid S \ x \Rightarrow N_2[x] \text{ end} \longrightarrow N_2[M] \quad (\text{R-MATCHS})$$

$$\text{if true then } N_1 \text{ else } N_2 \longrightarrow N_1 \quad (\text{R-IFT})$$

$$\text{if false then } N_1 \text{ else } N_2 \longrightarrow N_2 \quad (\text{R-IFF})$$

どちらの構文についてもふたつの規則が用意されていて場合分けに沿っていることがわかる. また, 規則 R-MATCHS では, 場合分けの対象である $S \ M$ の前の数 (すなわち M) が変数に代入されている.

最後は再帰関数の呼出しを表現した規則である. これは β 簡約とほぼ同じだが再帰を表現するために, 実引数だけでなく再帰関数そのものが再帰的参照である x に代入される.

$$(\text{fix } x(y : S) : T := M[x, y]) \ N \longrightarrow M[\text{fix } x(y : S) : T := M[x, y], N] \quad (\text{R-FIX})$$

部分項の簡約を許すための規則は以下のようなになる.

$$\frac{M \longrightarrow M'}{M \ N \longrightarrow M' \ N} \quad (\text{RC-APP1})$$

$$\frac{N \longrightarrow N'}{M \ N \longrightarrow M \ N'} \quad (\text{RC-APP2})$$

$$\frac{M \longrightarrow M'}{\text{match } M \text{ with } 0 \Rightarrow N_1 \mid S \ x \Rightarrow N_2 \text{ end} \longrightarrow \text{match } M' \text{ with } 0 \Rightarrow N_1 \mid S \ x \Rightarrow N_2 \text{ end}} \quad (\text{RC-MATCH1})$$

$$\frac{N_1 \longrightarrow N'_1}{\text{match } M \text{ with } 0 \Rightarrow N_1 \mid S \ x \Rightarrow N_2 \text{ end} \longrightarrow \text{match } M \text{ with } 0 \Rightarrow N'_1 \mid S \ x \Rightarrow N_2 \text{ end}} \quad (\text{RC-MATCH2})$$

$$\frac{N_2 \longrightarrow N'_2}{\text{match } M \text{ with } 0 \Rightarrow N_1 \mid S \ x \Rightarrow N_2 \text{ end} \longrightarrow \text{match } M \text{ with } 0 \Rightarrow N_1 \mid S \ x \Rightarrow N'_2 \text{ end}} \quad (\text{RC-MATCH3})$$

$$\frac{M \longrightarrow M'}{\text{if } M \text{ then } N_1 \text{ else } N_2 \longrightarrow \text{if } M' \text{ then } N_1 \text{ else } N_2} \quad (\text{RC-IF1})$$

$$\frac{N_1 \longrightarrow N'_1}{\text{if } M \text{ then } N_1 \text{ else } N_2 \longrightarrow \text{if } M \text{ then } N'_1 \text{ else } N_2} \quad (\text{RC-IF2})$$

$$\frac{N_2 \longrightarrow N'_2}{\text{if } M \text{ then } N_1 \text{ else } N_2 \longrightarrow \text{if } M \text{ then } N_1 \text{ else } N'_2} \quad (\text{RC-IF3})$$

$$\frac{M \longrightarrow M'}{\text{fun } x : T \Rightarrow M \longrightarrow \text{fun } x : T \Rightarrow M'} \quad (\text{RC-FUN})$$

$$\frac{M \longrightarrow M'}{\text{fix } x(y : S) : T := M \longrightarrow \text{fix } x(y : S) : T := M'} \quad (\text{RC-FIX})$$

最後に、複数ステップ (0 ステップ以上) での簡約を表す $M \longrightarrow^* N$ と、簡約を通じて二項が「等しい」ことを示す⁶ $M \longleftrightarrow N$ を規則の形で定義する。

$$\begin{array}{l} \frac{}{M \longleftrightarrow M} \quad (\text{EQ-REFL}) \\ \frac{}{M \longrightarrow^* M} \quad (\text{MR-ZERO}) \\ \frac{M \longrightarrow M'}{M \longrightarrow^* M'} \quad (\text{MR-ONE}) \\ \frac{M \longrightarrow^* M' \quad M' \longrightarrow^* M''}{M \longrightarrow^* M''} \quad (\text{MR-TRANS}) \\ \frac{M_1 \longleftrightarrow M_2 \quad M_2 \longleftrightarrow M_3}{M_1 \longleftrightarrow M_3} \quad (\text{EQ-TRANS}) \end{array}$$

$$\begin{array}{l} \frac{}{M \longleftrightarrow M} \quad (\text{EQ-REFL}) \\ \frac{M \longrightarrow N}{M \longleftrightarrow N} \quad (\text{EQ-RED}) \\ \frac{M \longleftrightarrow N}{N \longleftrightarrow M} \quad (\text{EQ-SYM}) \end{array}$$

$M \longleftrightarrow N$ は直感的には、 M から N に複数ステップの簡約 (ただし簡約の向きは両方向混ざっていてもよい) で到達することを表しており、Coq において項が等しいことを表す基本概念となっている。reflexivity を使って証明が完了するのは両辺が $M \longleftrightarrow N$ で関係づけられる時である。

練習問題 3.1 項

$M = \text{if true then (fun n : nat => plus n n) (S 0) else (fun n : nat => n) 0}$

とする。 M は以下の 3 つの項

$M_1 = \text{(fun n : nat => plus n n) (S 0)}$

$M_2 = \text{if true then plus (S 0) (S 0) else (fun n : nat => n) 0}$

$M_3 = \text{if true then (fun n : nat => plus n n) (S 0) else 0}$

に簡約されうるが、 $M \longrightarrow M_i$ (ただし $i = 1, 2, 3$) の導出木をそれぞれ書け。

練習問題 3.2 ラムダ計算での関数 (fun) は全て一引数関数である。しかし、fun を入れ子にすることで、複数引数関数を表現することができる。このアイデアについて説明せよ。

練習問題 3.3 Basics.v に登場した plus 関数を fix を使って表し、plus (S 0) (S 0) が簡約されて S (S 0) になる過程を、plus (S 0) (S 0) $\longrightarrow M_1 \longrightarrow \dots \longrightarrow M_n \longrightarrow S (S 0)$ なる M_i を列挙することで示せ。

⁶β 同値関係とも呼ばれる

プログラムの実行と簡約: 簡約は、基本的にはプログラムの実行と対応づけられる概念と言えるが、いくつか、ふつうのプログラムの実行とは異なる部分もある。

まず、R-BETA をよく見ると、多くのプログラミング言語での関数呼出しと異なり、引数 N は任意の項が許されており、引数の計算が終わっていないうちに関数を呼出すようなことも可能である。また、RC-FUN, RC-FIX 規則は、(再帰) 関数の本体も簡約してよいことを示している。実際、

```
Example foo : (fun (x:nat) : nat => 0 + x) = (fun (x:nat) : nat => x).
```

が単純化だけで証明できる。

これらの点は、簡約をプログラムの「実行」だと考えると、不自然かもしれないが、関数のインライン展開などのコンパイラの最適化のようなものまで考慮して「(同じことをする) より簡単なプログラム」を関係づけている、と思えばよいのではないだろうか。

変数のキャプチャ: 上の例のような、関数本体の計算をしてよいことを考えると、関数を true , $S\ 0$ のような具体的な値ではなく、変数(を含む式)で呼ぶことも許されていることがわかる。 β 簡約を素朴にパラメータを実引数で置き換えることだと思ってしまうと妙なことが起こる。例えば、 f を $\text{fun (x:nat) => fun (y:nat) => y + x}$ とし、 $\text{fun (y:nat) => f (y*y) 0}$ を簡約することを考えよう。直観的には、 f は単なる足し算を(引数の順をひっくり返して)行うだけであるから、 $\text{fun (y:nat) => 0 + y * y}$ が得られるはずである。しかし、最初のステップとして $f (y*y)$ の部分の β 簡約を、関数本体の

$$\text{fun (y:nat) => y + x}$$

中の x を $y*y$ で置き換える作業と思うと、

$$\text{fun (y:nat) => y + y * y}$$

が得られ、次の 0 を渡すことで、 $\text{fun (y:nat) => 0 + 0 * 0}$ になってしまう。

これは f のパラメータである y と、呼出し側にある $y * y$ の y を混同してしまったためである。ラムダ計算では、このような混同が起こる場合には、関数パラメータの名前換えをして混同を避ける。上の例では、 f を $\text{fun (x:nat) => fun (y0:nat) => y0 + x}$ とした上で β 簡約を行えば、

$$\text{fun (y0:nat) => y0 + y * y}$$

が得られて、これを 0 で呼べば

$$0 + y * y$$

が得られる。

ラムダ計算では、引数の名前替えをした項同士は同一視するのが慣習である。同様のことは、他の変数宣言を伴う構文についてもいえる。

3.3 簡約に関する重要な性質

ラムダ計算の簡約に関する重要な性質は、計算はどこから手をつけても結果は変わらないことを示す、合流性 (*confluence*) である。合流性は以下のような定理として述べることができる。

定理 1 (合流性) 任意の項 M_1, M_2, M_3 に対して, $M_1 \rightarrow^* M_2$ かつ $M_1 \rightarrow^* M_3$ ならば, ある M_4 が存在し, $M_2 \rightarrow^* M_4$ かつ $M_3 \rightarrow^* M_4$ が成り立つ.

ある項の簡約を進め, 簡約がこれ以上できない形になる場合, その最終項を正規形 (normal form) という. `Compute` コマンドは, 項の正規形を求めるためのコマンドである. 合流性が成立すると, 項に対する正規形が高々ひとつであることがわかる.

練習問題 3.4 合流性が成立するならば, 項に対する正規形が高々ひとつであることを説明せよ.

3.4 型付け関係

項 M の型が T であることを $M : T$ と書く. 例えば `S 0 : nat` や

```
fun n:nat => match n with 0 => true | S n' => false end : nat -> bool
```

である. Coq であれば項の型は `Check` コマンドを使って知ることができるが, この項と型の関係 (これを型付け関係という) の正確な定義をみていく.

項には変数が現れるため, 一般には型付け関係は変数の型についての情報 Γ を加えた三項関係 $\Gamma \vdash M : T$ として表す. Γ は, $x : T$ という形の変数の型宣言の列であり, 文脈 (context) または型環境 (type environment)⁷ と呼ばれる. 例えば,

```
x:nat ⊢ S x : nat
n:nat, b:bool ⊢ if b then n else S n : nat
n:nat ⊢ fun b:bool => if b then n else S n : bool -> nat
```

といった関係が成立する.

型付け関係も簡約と同様に規則を使って定義することができる. 型付け関係のための規則は型付け規則 (typing rule) と呼ばれる. 以下が, 型付け規則である.

$$\frac{(x : T \in \Gamma)}{\Gamma \vdash x : T} \quad (\text{T-VAR})$$

$$\frac{}{\Gamma \vdash 0 : \text{nat}} \quad (\text{T-ZERO})$$

$$\frac{}{\Gamma \vdash S : \text{nat} \rightarrow \text{nat}} \quad (\text{T-SUCC})$$

$$\frac{\Gamma \vdash M : \text{nat} \quad \Gamma \vdash N_1 : T \quad \Gamma, x : \text{nat} \vdash N_2 : T \quad (x \notin \text{dom}(\Gamma))}{\Gamma \vdash \text{match } M \text{ with } 0 \Rightarrow N_1 \mid S x \Rightarrow N_2 \text{ end} : T} \quad (\text{T-MATCH})$$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \quad (\text{T-TRUE})$$

⁷正確には

$$\Gamma ::= \bullet \\ \quad \mid \Gamma, x : T$$

という構文で定義される. 先頭の \bullet は省略することが多い.

$$\frac{}{\Gamma \vdash \text{false} : \text{bool}} \quad (\text{T-FALSE})$$

$$\frac{\Gamma \vdash M : \text{bool} \quad \Gamma \vdash N_1 : T \quad \Gamma \vdash N_2 : T}{\Gamma \vdash \text{if } M \text{ then } N_1 \text{ else } N_2 : T} \quad (\text{T-IF})$$

$$\frac{\Gamma, x : S \vdash M : T \quad (x \notin \text{dom}(\Gamma))}{\Gamma \vdash \text{fun } x : S \Rightarrow M : S \rightarrow T} \quad (\text{T-FUN})$$

$$\frac{\Gamma, x : S \rightarrow T, y : S \vdash M : T \quad (x \notin \text{dom}(\Gamma)) \quad (y \notin \text{dom}(\Gamma))}{\Gamma \vdash \text{fix } x(y : S) : T := M : S \rightarrow T} \quad (\text{T-FIX})$$

$$\frac{\Gamma \vdash M : S \rightarrow T \quad \Gamma \vdash N : S}{\Gamma \vdash M N : T} \quad (\text{T-APP})$$

- 規則 T-MATCH, T-IF では、分岐後の項の型が等しいことを要求している。
- 規則 T-FUN によると、関数に型 $S \rightarrow T$ がつくのは、パラメータの型を S として (文脈に追加して)、本体式に T 型がつく時であることがわかる。再帰関数についても、 x の型が全体の型と等しくなることに注意すれば、ほぼ同様である。

$\text{dom}(\Gamma)$ は Γ 中のコロンの左側に現れる変数からなる集合である。これは、文脈に既に現れている変数は文脈に追加できないことを示している。このため、 $\text{fun } x : \text{nat} \Rightarrow \text{fun } x : \text{bool} \Rightarrow x$ には型がつかないように思えるが、これも変数のキャプチャのと同じアイデアを使ってパラメータの名前替えを行い、 $\text{fun } x : \text{nat} \Rightarrow \text{fun } x_0 : \text{bool} \Rightarrow x_0$ と考える⁸ことで回避できる。

型付け関係も (簡約と同様に) 導出木を使って、その関係がなぜ成立するのかを説明することができる。例えば $\vdash \text{fun } n : \text{nat} \Rightarrow \text{match } n \text{ with } 0 \Rightarrow \text{true} \mid S \ n' \Rightarrow \text{false} \text{ end} : \text{nat} \rightarrow \text{bool}$ の導出木は以下のようなになる。(以下 $\Gamma = n : \text{nat}$ とする。)

$$\frac{\frac{\Gamma \vdash n : \text{nat}}{\Gamma \vdash n : \text{nat}} \text{T-VAR} \quad \frac{}{\Gamma \vdash \text{true} : \text{bool}} \text{T-TRUE} \quad \frac{}{\Gamma, n' : \text{nat} \vdash \text{true} : \text{bool}} \text{T-FALSE}}{\Gamma \vdash \text{match } n \text{ with } 0 \Rightarrow \text{true} \mid S \ n' \Rightarrow \text{false} \text{ end} : \text{bool}} \text{T-MATCH} \quad \text{T-FUN}$$

練習問題 3.5 Basics.v に登場した plus 関数を fix を使って表し (M とする), 型付け関係

$$\vdash M : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$$

の導出木を書け。

⁸ $\text{fun } x_0 : \text{nat} \Rightarrow \text{fun } x : \text{bool} \Rightarrow x_0$ ではない, ことに注意. 変数の使用は一番近い宣言を参照する.

3.5 型付けに関する重要な性質

型を使うことの恩恵のひとつは「意味のないプログラム」を排除することができることにある。「意味のないプログラム」とは、データ・関数に対して想定されていない使い方をするような項で

- 0 true のような、関数ではない値の適用
- $\text{if } (\text{fun } x : \text{nat} \Rightarrow \dots) \text{ then } \dots \text{ else } \dots$ や真偽値以外での場合わけ

が簡約の過程で発生するような項ということが出来る。

単純型付ラムダ計算については、以下の「型保存定理」と「前進性」という定理が成立する。

定理 2 (型保存定理) $\Gamma \vdash M : T$ かつ、 $M \longrightarrow M'$ ならば、 $\Gamma \vdash M' : T$ である。

項 M が標準形 (*canonical form*) である、とは、 M が $0, S N, \text{true}, \text{false}, \text{fun } x : T \Rightarrow N$, もしくは、 $\text{fix } x(y : S) : T := N$ いずれかの形をしていることをいう。

定理 3 (前進性) $\vdash M : T$ とする。任意の M の部分項 M_0 について、以下が成立する。

- M_0 が $M_1 M_2$ の形 (ただし M_1 は標準形) ならば、 M_1 は $\text{fun } x : T' \Rightarrow M'$ もしくは $\text{fix } x(y : S') : T' := M'$ の形である。
- M_0 が $\text{match } M_1 \text{ with } 0 \Rightarrow M_2 \mid S x \Rightarrow M_3 \text{ end}$ の形 (ただし M_1 は標準形) ならば、 M_1 は 0 または、 $S M'$ の形である。
- M_1 が $\text{if } M_1 \text{ then } M_2 \text{ else } M_3$ の形 (ただし M_1 は標準形) ならば、 M_1 は true もしくは false である。

前進性は、間接的な言い方ではあるが、簡約が進みそうな項 (標準形が現れる関数適用, *match*, *if*) は実際に簡約が進む (0 true のようなエラーと考えられる項ではない) ことを示している。

これらのふたつの定理を合わせると、空の文脈の下で型を持つ項について、もし簡約が停止するならば、その結果得られた正規形は、 $S(\dots(0)\dots)$, true , false , $\text{fun } x : T \Rightarrow N$, もしくは、 $\text{fix } x(y : S) : T := N$ いずれかの形をしていることがわかる。

さらに、明示的に再帰を使わない限り、型のついた項の簡約が止まる・発散することはないと言える。これを正規化性という。正規化性には弱・強の2種類が存在し、微妙に内容が違う (強正規化性は弱正規化性を含意するが逆は成り立たない) が単純型付ラムダ計算ではどちらも成立する。

定理 4 (正規化性) M には *fix* が現れず、かつ、 $\Gamma \vdash M : T$ とする。

1. $M \longrightarrow^* N$ かつ、 N はそれ以上は簡約できないような N が存在する。(弱正規化性)
2. $M \longrightarrow M_1 \longrightarrow M_2 \longrightarrow \dots$ なる項の無限列 M_1, M_2, \dots は存在しない。(強正規化性)

Coq では、*fix* を使っている強正規化性を持つような項しか許されていない。(型とは別の方法で止まることが検査されている。) 強正規化性と合流性が成立すると、 $M \longleftrightarrow N$ を判定する問題が決定可能になる。

練習問題 3.6 強正規化性と合流性が成立すると、 $M \longleftrightarrow N$ を判定する問題が決定可能になるのはなぜか説明せよ。