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
cal20@fos.kuis.kyoto-u.ac.jp
http://www.fos.kuis.kyoto-u.ac.jp/~igarashi/class/cal/

October 20, 2020

`Compute` によるプログラムの実行，タクティック `simpl` による式の単純化がどのようになされるか，また `reflexivity` を使って証明が成功する条件 (すなわち，=の両辺が「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ための理論的枠組みのことで，プログラム (を一般化した項 (term) と呼ばれるもの) の変形過程を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

まず準備運動として `S`, `O` 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

項の集合 *Terms* は BNF を使って以下のように定義される．

$$\begin{array}{rlcl} \text{(terms)} & \boldsymbol{M}, \boldsymbol{N} \in \mathit{Terms} & ::= & \mathtt{O} \\ & & \mid & \mathtt{S}\ \boldsymbol{M} \\ & & \mid & \boldsymbol{M} + \boldsymbol{N} \\ & & \mid & \boldsymbol{M} * \boldsymbol{N} \end{array}$$

ゼロ，サクセサ (successor, 次の数)，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 O` は `S O` に `S` をかぶせたものとしか読みようがないのだ

[1] $\boldsymbol{M} + \boldsymbol{N} * \boldsymbol{P}$ は $\boldsymbol{M} + (\boldsymbol{N} * \boldsymbol{P})$ を示す．
[2] $\boldsymbol{M} + \boldsymbol{N} + \boldsymbol{P}$ は，$(\boldsymbol{M} + \boldsymbol{N}) + \boldsymbol{P}$ を，$\boldsymbol{M} * \boldsymbol{N} * \boldsymbol{P}$ は，$(\boldsymbol{M} * \boldsymbol{N}) * \boldsymbol{P}$ を，示す．

が，後で行われる項の拡張を考えて，`S (S O)` と書くことにする．また，`S` $\boldsymbol{M}$ は `+`, `*` よりも強く結合[3]する．

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，`O + S O` が `S O` になる，といったことが考えられる．これを

$$\texttt{O + S O} \longrightarrow \texttt{S O}$$
$$\texttt{S O * (O + S O)} \longrightarrow \texttt{S O * S O}$$

のように，矢印を使って表す．数学的には，集合 *Terms* 上の二項関係 $\longrightarrow \subseteq \mathit{Terms} \times \mathit{Terms}$ を導入することになる．ふたつの項 $\boldsymbol{M}, \boldsymbol{N}$ がこの関係で関係付けられている，つまり，対 $(\boldsymbol{M}, \boldsymbol{N})$ がこの集合の元である $((\boldsymbol{M}, \boldsymbol{N}) \in \longrightarrow)$ ことを $\boldsymbol{M} \longrightarrow \boldsymbol{N}$ と書く．直感的な意味は「$\boldsymbol{M}$ が 1 ステップで $\boldsymbol{N}$ に簡約される」ということである．簡約関係の左辺で，変形される部分項を**簡約基** (*redex*)(reducible expression の略) という．

簡約関係は**推論規則** (*inference rule*) と呼ばれる形式を使って与えられる．以下は，「どんな項 $\boldsymbol{M}$ に対しても，`O +` $\boldsymbol{M} \longrightarrow \boldsymbol{M}$ という関係が成立する」ことを示す推論規則である．

$$\texttt{O +}\ \boldsymbol{M} \longrightarrow \boldsymbol{M} \qquad \text{(R-PlusZ)}$$

右端に書かれた R-PlusZ はこの推論規則の名前である．この推論規則は，$\boldsymbol{M}$ 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，$\boldsymbol{M} =$ `S O` とすれば，

$$\texttt{O + S O} \longrightarrow \texttt{S O}$$

が導けるし，$\boldsymbol{M} =$ `S (S (S O))` とすれば，

$$\texttt{O + S (S (S O))} \longrightarrow \texttt{S (S (S O))}$$

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` の定義との対応を見てとってほしい．

$$\texttt{(S}\ \boldsymbol{M}\texttt{)}\ \texttt{+}\ \boldsymbol{N} \longrightarrow \texttt{S (}\boldsymbol{M}\ \texttt{+}\ \boldsymbol{N}\texttt{)} \qquad \text{(R-PlusS)}$$

$$\texttt{O *}\ \boldsymbol{M} \longrightarrow \texttt{O} \qquad \text{(R-MultZ)}$$

$$\texttt{(S}\ \boldsymbol{M}\texttt{)}\ \texttt{*}\ \boldsymbol{N} \longrightarrow \boldsymbol{N}\ \texttt{+}\ \boldsymbol{M}\ \texttt{*}\ \boldsymbol{N} \qquad \text{(R-MultS)}$$

これらの規則を使うと，

$$\texttt{S O + S O} \longrightarrow \texttt{S (O + S O)}$$

や

$$\texttt{S O * S O} \longrightarrow \texttt{S O + O * S O}$$

[3] `S` $\boldsymbol{M}$`+`$\boldsymbol{N}$ は `(S` $\boldsymbol{M}$`)+`$\boldsymbol{N}$ を示す．

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自然に想定される

$$\mathtt{S(O\ +\ S\ O)} \longrightarrow \mathtt{S(S\ O)}$$

や

$$\mathtt{S\ O\ +\ O\ *\ S\ O} \longrightarrow \mathtt{S\ O\ +\ O}$$

のように，項の一部分 (**部分項** (*subterm*) 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．

$$\frac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\mathtt{S}\ \boldsymbol{M} \longrightarrow \mathtt{S}\ \boldsymbol{M'}} \qquad \text{(RC-Succ)}$$

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 $\boldsymbol{M} \longrightarrow \boldsymbol{N}$ 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，

$$\begin{aligned} \boldsymbol{M} &= \mathtt{O\ +\ S\ O} \\ \boldsymbol{M'} &= \mathtt{S\ O} \end{aligned}$$

とすると，この規則は，

$\mathtt{O\ +\ S\ O} \longrightarrow \mathtt{S\ O}$ がいえたなら
$\mathtt{S(O\ +\ S\ O)} \longrightarrow \mathtt{S\ (S\ O)}$ をいってもよい

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,「$\mathtt{S(O\ +\ S\ O)} \longrightarrow \mathtt{S\ (S\ O)}$ をいってもよい」になる．このような，

1. 規則 R-PlusZ より (1) $\mathtt{O\ +\ S\ O} \longrightarrow \mathtt{S\ O}$
2. 上の関係 (1) と規則 RC-Succ より $\mathtt{S\ (O\ +\ S\ O)} \longrightarrow \mathtt{S\ (S\ O)}$

という，関係を確認するための推論過程を

$$\dfrac{\dfrac{}{\mathtt{O\ +\ S\ O} \longrightarrow \mathtt{S\ O}}\ \text{R-PlusZ}}{\mathtt{S\ (O\ +\ S\ O)} \longrightarrow \mathtt{S\ (S\ O)}}\ \text{RC-Succ}$$

という推論規則を縦に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** (derivation tree)，または単に**導出** (derivation) と呼ぶ．

この規則 RC-Succ は，導出関係にある二項それぞれの外側に S をつけても簡約関係にある，別の言い方をすると，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，

簡約基はどれを先に選んで計算してもよい

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になる．

$$\frac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\mathtt{S}\ \boldsymbol{M} \longrightarrow \mathtt{S}\ \boldsymbol{M'}} \quad \text{(RC-Succ)}$$

$$\frac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\boldsymbol{M}\ \mathtt{+}\ \boldsymbol{N} \longrightarrow \boldsymbol{M'}\ \mathtt{+}\ \boldsymbol{N}} \quad \text{(RC-PlusL)}$$

$$\frac{\boldsymbol{N} \longrightarrow \boldsymbol{N'}}{\boldsymbol{M}\ \mathtt{+}\ \boldsymbol{N} \longrightarrow \boldsymbol{M}\ \mathtt{+}\ \boldsymbol{N'}} \quad \text{(RC-PlusR)}$$

$$\frac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\boldsymbol{M}\ \mathtt{*}\ \boldsymbol{N} \longrightarrow \boldsymbol{M'}\ \mathtt{*}\ \boldsymbol{N}} \quad \text{(RC-MultL)}$$

$$\frac{\boldsymbol{N} \longrightarrow \boldsymbol{N'}}{\boldsymbol{M}\ \mathtt{*}\ \boldsymbol{N} \longrightarrow \boldsymbol{M}\ \mathtt{*}\ \boldsymbol{N'}} \quad \text{(RC-MultR)}$$

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

- $\boldsymbol{M_1} = \mathtt{S\ 0\ +\ (S\ 0\ *\ S\ 0)}$
- $\boldsymbol{M_2} = \mathtt{(S\ 0\ +\ S\ 0)\ *\ (S\ (S\ 0)\ +\ S\ 0)}$
- $\boldsymbol{M_3} = \mathtt{S\ (S\ (S\ (S\ 0)))\ +\ 0}$

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

1.3 マルチステップ簡約，簡約に基づく項の等しさ

上で導入した $\longrightarrow$ をもとに，さらにふたつの関係 $\longrightarrow^*$ と $\longleftrightarrow$ を規則を使って定義する．$\boldsymbol{M} \longrightarrow^* \boldsymbol{N}$ は「$\boldsymbol{M}$ を 0 回以上 (有限回) 簡約すると $\boldsymbol{N}$ が得られる」という意味の関係，$\boldsymbol{M} \longleftrightarrow \boldsymbol{N}$ は「$\boldsymbol{M}$ から，向きはともかく簡約を (有限回) 繰り返すと $\boldsymbol{N}$ が得られる」という意味の関係である．(「向きはともかく」なので Eq-Symm で表される対称律が入っている．)

$$\boldsymbol{M} \longrightarrow^* \boldsymbol{M} \quad \text{(MR-Refl)}$$

$$\frac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\boldsymbol{M} \longrightarrow^* \boldsymbol{M'}} \quad \text{(MR-One)}$$

$$\frac{\boldsymbol{M} \longrightarrow^* \boldsymbol{M'} \qquad \boldsymbol{M'} \longrightarrow^* \boldsymbol{M''}}{\boldsymbol{M} \longrightarrow^* \boldsymbol{M''}} \quad \text{(MR-Trans)}$$

$$\boldsymbol{M} \longleftrightarrow \boldsymbol{M} \quad \text{(Eq-Refl)}$$

$$\frac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\boldsymbol{M} \longleftrightarrow \boldsymbol{M'}} \quad \text{(Eq-One)}$$

$$\frac{\boldsymbol{M} \longleftrightarrow \boldsymbol{M'}}{\boldsymbol{M'} \longleftrightarrow \boldsymbol{M}} \quad \text{(Eq-Symm)}$$

$$\frac{\boldsymbol{M} \longleftrightarrow \boldsymbol{M'} \qquad \boldsymbol{M'} \longleftrightarrow \boldsymbol{M''}}{\boldsymbol{M} \longleftrightarrow \boldsymbol{M''}} \quad \text{(Eq-Trans)}$$

例えば，`S O + S O` $\longrightarrow^*$ `S (S O)` や，`O + S (S O)` $\longleftrightarrow$ `S O + S O` といった関係が成立する．

練習問題 1.3 `S O + S O` $\longrightarrow^*$ `S (S O)` の導出木を書け．

練習問題 1.4 `S (S O) + O` $\longleftrightarrow$ `O + S (S O)` の導出木を書け．

$\longrightarrow^*$ は MR-Refl や MR-Trans からわかるように反射的かつ推移的な関係である．より正確には $\longrightarrow$ を含む最小の反射的推移的な関係，すなわち，$\longrightarrow$ の**反射的推移的閉包** (*reflexive transtivie closure*) である．また，$\longleftrightarrow$ は Eq-Refl, Eq-Symm，Eq-Trans からわかるように反射的，対称的，かつ推移的な関係，つまり**同値関係** (*equivalence relation*) である．

$\longrightarrow^*$ は Coq での `Compute` や `simpl` による式の単純化をモデル化した関係である．また，$\longleftrightarrow$ は `reflexivity` で「等しい」と判断される項の関係をモデル化している．定義からは直接は読み取れないが，実は，$\boldsymbol{M} \longleftrightarrow \boldsymbol{N}$ であることと，$\boldsymbol{M}$ と $\boldsymbol{N}$ の計算結果 (簡約がこれ以上進まないところまで進めた項，**正規形** (normal form) という) が同じ項であることは同値である．

メタ変数とオブジェクト変数　ここで扱った項は具体的な自然数についての加算・乗算を表している．簡約規則を表すためには $\boldsymbol{M}$ といった，項一般を表すための「変数」を使っているが，これはあくまで日本語での説明のために導入した記号である．我々は，日本語を使って，具体的な自然数についての加算・乗算式の言語を説明しているわけだが，説明に使う言語を**メタ言語** (meta language)，説明されている言語を**対象言語** (object language) という．$\boldsymbol{M}$ のようなメタ言語から対象言語の「もの」を指し示すために使う変数を**メタ変数** metavariable という．一方，Coq のプログラムにおいては，関数のパラメータを表すために `n`, `b` といった変数を使うが，これは対象言語に属する変数である (メタ変数と区別するために，プログラム変数，対象言語の変数，などということもある)．

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「(入力と出力を表す) 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，$\lambda\langle$ パラメータ $\rangle.\langle$ 式 $\rangle$ 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実引数を与えた時 (関数を適用する，という)，関数の値は「パラメータを実引数で具体化 (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，

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

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．

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

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

上の例での

$$(\lambda x.x^2\pi)(2) + f(7) + f(20) \quad \longrightarrow \quad 2^2\pi + f(7) + f(20)$$

というステップが β 簡約の例になっている．

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 (untyped λ-calculus)」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*) ということも) 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．

[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 BNF で定義する．

$$
\begin{array}{llll}
\text{(types)} & \boldsymbol{S}, \boldsymbol{T} & ::= & \texttt{nat} \\
& & | & \texttt{bool} \\
& & | & \boldsymbol{S}\ \texttt{->}\ \boldsymbol{T} \\
\\
\text{(variables)} & \boldsymbol{x}, \boldsymbol{y} & \in & \{\texttt{a}, \texttt{b}, \ldots\} \\
\text{(terms)} & \boldsymbol{M}, \boldsymbol{N} & ::= & \boldsymbol{x} \\
& & | & \texttt{O} \\
& & | & \texttt{S} \\
& & | & \texttt{match}\ \boldsymbol{M}\ \texttt{with O =>}\ \boldsymbol{N_1}\ \texttt{| S}\ \boldsymbol{x}\ \texttt{=>}\ \boldsymbol{N_2}\ \texttt{end} \\
& & | & \texttt{true} \\
& & | & \texttt{false} \\
& & | & \texttt{if}\ \boldsymbol{M}\ \texttt{then}\ \boldsymbol{N_1}\ \texttt{else}\ \boldsymbol{N_2} \\
& & | & \texttt{fun}\ \boldsymbol{x} : \boldsymbol{T}\ \texttt{=>}\ \boldsymbol{M} \\
& & | & \texttt{fix}\ \boldsymbol{x}\texttt{(}\boldsymbol{y} : \boldsymbol{S}\texttt{)} : \boldsymbol{T}\ \texttt{:=}\ \boldsymbol{M} \\
& & | & \boldsymbol{M_1}\ \boldsymbol{M_2}
\end{array}
$$

- 型としては，自然数の型 `nat`，真偽値の型 `bool`，と関数型 $\boldsymbol{S}$ `->` $\boldsymbol{T}$ を考える．この $\boldsymbol{S}$ を引数型，$\boldsymbol{T}$ を返値型ということがある．`->` は右結合する．例えば，$\boldsymbol{T_1}$ `->` $\boldsymbol{T_2}$ `->` $\boldsymbol{T_3}$ は $\boldsymbol{T_1}$ `->` ($\boldsymbol{T_2}$ `->` $\boldsymbol{T_3}$) のことであり，($\boldsymbol{T_1}$ `->` $\boldsymbol{T_2}$) `->` $\boldsymbol{T_3}$ 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 (これまでの構文で書くなら `match` $\boldsymbol{M_1}$ `with true =>` $\boldsymbol{M_2}$ `| false =>` $\boldsymbol{M_3}$ `end` と書かれるもの)，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，$\lambda \boldsymbol{x} : \boldsymbol{T}.\boldsymbol{M}$ 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 `fix` $\boldsymbol{x}$`(`$\boldsymbol{y}$ `:` $\boldsymbol{S}$`) :` $\boldsymbol{T}$ `:=` $\boldsymbol{M}$ は Coq の `Fixpoint` で定義される関数に相当する．$\boldsymbol{x}$ が関数 (を再帰的に参照するため) の名前，$\boldsymbol{y}$ がパラメータ，$\boldsymbol{S}$ がパラメータの型，$\boldsymbol{T}$ が関数本体 $\boldsymbol{M}$ 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

前と同じく，計算過程を表す簡約関係は $\boldsymbol{M} \longrightarrow \boldsymbol{N}$ という形で書かれ,「$\boldsymbol{M}$ が 1 ステップで $\boldsymbol{N}$ 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，先程述べたように

$$\texttt{(fun } \boldsymbol{x} \texttt{ : } \boldsymbol{T} \texttt{ => } \boldsymbol{M}[\boldsymbol{x}]\texttt{)} \ \boldsymbol{N} \longrightarrow \boldsymbol{M}[\boldsymbol{N}] \qquad \text{(R-Beta)}$$

で表される．R-Beta を使うと，例えば，

$$\begin{aligned} \boldsymbol{x} &= \texttt{n} \\ \boldsymbol{T} &= \texttt{nat} \\ \boldsymbol{M}[\boldsymbol{x}] &= \texttt{match } \boldsymbol{x} \texttt{ with O => S O | S n' => S (S } \boldsymbol{x}\texttt{) end} \\ \boldsymbol{N} &= \texttt{S O} \end{aligned}$$

とすれば，具体的な二項間の関係

```
(fun n : nat => match n with O => S O | S n' => S (S n) end) (S O)
          ⟶ match S O with O => S O | S n' => S (S (S O)) 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．

$$\texttt{match O with O => } \boldsymbol{N_1} \texttt{ | S } \boldsymbol{x} \texttt{ => } \boldsymbol{N_2} \texttt{ end} \longrightarrow \boldsymbol{N_1} \tag{R-MatchZ}$$

$$\texttt{match S } \boldsymbol{M} \texttt{ with O => } \boldsymbol{N_1} \texttt{ | S } \boldsymbol{x} \texttt{ => } \boldsymbol{N_2}[\boldsymbol{x}] \texttt{ end} \longrightarrow \boldsymbol{N_2}[\boldsymbol{M}] \tag{R-MatchS}$$

$$\texttt{if true then } \boldsymbol{N_1} \texttt{ else } \boldsymbol{N_2} \longrightarrow \boldsymbol{N_1} \tag{R-IfT}$$

$$\texttt{if false then } \boldsymbol{N_1} \texttt{ else } \boldsymbol{N_2} \longrightarrow \boldsymbol{N_2} \tag{R-IfF}$$

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 $\boldsymbol{M}$ の前の数 (すなわち $\boldsymbol{M}$) 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 $\boldsymbol{x}$ に代入される．

$$\texttt{(fix } \boldsymbol{x}\texttt{(}\boldsymbol{y}\texttt{ : }\boldsymbol{S}\texttt{) : }\boldsymbol{T}\texttt{ := } \boldsymbol{M}[\boldsymbol{x},\boldsymbol{y}]\texttt{)} \; \boldsymbol{N} \longrightarrow \boldsymbol{M}[\texttt{fix } \boldsymbol{x}\texttt{(}\boldsymbol{y}\texttt{ : }\boldsymbol{S}\texttt{) : }\boldsymbol{T}\texttt{ := } \boldsymbol{M}[\boldsymbol{x},\boldsymbol{y}], \boldsymbol{N}] \tag{R-Fix}$$

部分項の簡約を許すための規則は以下のようになる．

$$\frac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\boldsymbol{M}\ \boldsymbol{N} \longrightarrow \boldsymbol{M'}\ \boldsymbol{N}} \tag{RC-App1}$$

$$\frac{\boldsymbol{N} \longrightarrow \boldsymbol{N'}}{\boldsymbol{M}\ \boldsymbol{N} \longrightarrow \boldsymbol{M}\ \boldsymbol{N'}} \tag{RC-App2}$$

$$\frac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\begin{array}{l}\texttt{match } \boldsymbol{M} \texttt{ with O => } \boldsymbol{N_1} \texttt{ | S } \boldsymbol{x} \texttt{ => } \boldsymbol{N_2} \texttt{ end} \\ \longrightarrow \texttt{match } \boldsymbol{M'} \texttt{ with O => } \boldsymbol{N_1} \texttt{ | S } \boldsymbol{x} \texttt{ => } \boldsymbol{N_2} \texttt{ end}\end{array}} \tag{RC-Match1}$$

$$\frac{\boldsymbol{N_1} \longrightarrow \boldsymbol{N_1'}}{\begin{array}{l}\texttt{match } \boldsymbol{M} \texttt{ with O => } \boldsymbol{N_1} \texttt{ | S } \boldsymbol{x} \texttt{ => } \boldsymbol{N_2} \texttt{ end} \\ \longrightarrow \texttt{match } \boldsymbol{M} \texttt{ with O => } \boldsymbol{N_1'} \texttt{ | S } \boldsymbol{x} \texttt{ => } \boldsymbol{N_2} \texttt{ end}\end{array}} \tag{RC-Match2}$$

$$\frac{\boldsymbol{N_2} \longrightarrow \boldsymbol{N_2'}}{\begin{array}{l}\texttt{match } \boldsymbol{M} \texttt{ with O => } \boldsymbol{N_1} \texttt{ | S } \boldsymbol{x} \texttt{ => } \boldsymbol{N_2} \texttt{ end} \\ \longrightarrow \texttt{match } \boldsymbol{M} \texttt{ with O => } \boldsymbol{N_1} \texttt{ | S } \boldsymbol{x} \texttt{ => } \boldsymbol{N_2'} \texttt{ end}\end{array}} \tag{RC-Match3}$$

$$\frac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\texttt{if } \boldsymbol{M} \texttt{ then } \boldsymbol{N_1} \texttt{ else } \boldsymbol{N_2} \longrightarrow \texttt{if } \boldsymbol{M'} \texttt{ then } \boldsymbol{N_1} \texttt{ else } \boldsymbol{N_2}} \tag{RC-If1}$$

$$\frac{\boldsymbol{N_1} \longrightarrow \boldsymbol{N_1'}}{\texttt{if } \boldsymbol{M} \texttt{ then } \boldsymbol{N_1} \texttt{ else } \boldsymbol{N_2} \longrightarrow \texttt{if } \boldsymbol{M} \texttt{ then } \boldsymbol{N_1'} \texttt{ else } \boldsymbol{N_2}} \tag{RC-If2}$$

$$\frac{N_2 \longrightarrow N_2'}{\texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2 \longrightarrow \texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2'} \quad \text{(RC-If3)}$$

$$\frac{M \longrightarrow M'}{\texttt{fun } x : T \texttt{ => } M \longrightarrow \texttt{fun } x : T \texttt{ => } M'} \quad \text{(RC-Fun)}$$

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

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]$M \longleftrightarrow N$ を規則を使って定義する．

$$\frac{}{M \longrightarrow^* M} \quad \text{(MR-Refl)}$$

$$\frac{M \longrightarrow M'}{M \longrightarrow^* M'} \quad \text{(MR-One)}$$

$$\frac{M \longrightarrow^* M' \qquad M' \longrightarrow^* M''}{M \longrightarrow^* M''} \quad \text{(MR-Trans)}$$

$$\frac{}{M \longleftrightarrow M} \quad \text{(Eq-Refl)}$$

$$\frac{M \longrightarrow N}{M \longleftrightarrow N} \quad \text{(Eq-Red)}$$

$$\frac{M \longleftrightarrow N}{N \longleftrightarrow M} \quad \text{(Eq-Sym)}$$

$$\frac{M_1 \longleftrightarrow M_2 \qquad M_2 \longleftrightarrow M_3}{M_1 \longleftrightarrow M_3} \quad \text{(Eq-Trans)}$$

この定義は，(もとになっている $\longrightarrow$ の意味こそ違うが) 自然数の加算と乗算での $\longrightarrow^*$ と $\longleftrightarrow$ を定義した規則と全く同じものである．$M \longleftrightarrow N$ は，Coq において項が計算を通じて等しいことを表す基本概念となっている．実際，ゴールが "M = N" の時，`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．(この関係が成立しているかを判定するアルゴリズムについては後述する．)

練習問題 3.1 項

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

とする．M は以下の 3 つの項

$$\begin{aligned} M_1 &= \texttt{(fun n : nat => plus n n) (S O)} \\ M_2 &= \texttt{if true then plus (S O) (S O) else (fun n : nat => n) O} \\ M_3 &= \texttt{if true then (fun n : nat => plus n n) (S O) else O} \end{aligned}$$

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．

[6] β 同値関係とも呼ばれる

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 O) (S O) が簡約されて S (S O) になる過程を，plus (S O) (S O) $\longrightarrow \boldsymbol{M_1} \longrightarrow \cdots \longrightarrow \boldsymbol{M_n} \longrightarrow$ S (S O) なる $\boldsymbol{M_i}$ を列挙することで示せ．

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 $\boldsymbol{N}$ として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．つまり，

```
(fun (x:nat) => match x with O => O | S y => y) (S(a + b))
```

(ここで a, b は変数) は，2 ステップで

```
a + b
```

に簡約される．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，

$$\texttt{(fun (x:nat) => 0 + x)} \longleftrightarrow \texttt{(fun (x:nat) => x)}$$

が成立し，

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

は reflexivity 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「(同じ計算をする) より簡単なプログラム」と関係づけている，と思えばよい．

代入と変数の「捕捉」: 上の例のように，関数を true, S O 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．β 簡約を素朴にパラメータを実引数で**置き換える**ことだと思うと妙なことが起こる．例えば，f を fun x:nat => fun y:nat => y + x と定義し，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，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

```
fun y:nat => y + x
```

中の x を y*y で置き換える作業と思うと，

```
fun y:nat => y + y * y
```

が得られ，次の 0 を渡すことで，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 fun x:nat => fun (y0:nat) => y0 + x とした上で β 簡約を行えば，

```
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．

これを確認するために，Coq で

```
Definition f (x y : nat) => y + x.
Eval compute [f] in (fun y:nat => f (y*y)).
```

を実行してみよ．(`Eval compute [f]` は定義の展開を `f` に限って行う (+の定義は展開しない) ためのコマンドである．0 を抜いているのは，0 での呼び出しをせずに止める方法がわからないためである．)

このような変数の捕捉を回避しながら行う代入を capture-avoiding substitution と呼ぶ．また，ラムダ計算では，変数の名前替えをした前後の項同士は同一視するのが慣習である．同様のことは，`match` など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 $\boldsymbol{M_1}$, $\boldsymbol{M_2}$, $\boldsymbol{M_3}$ に対して，$\boldsymbol{M_1} \longrightarrow^* \boldsymbol{M_2}$ かつ $\boldsymbol{M_1} \longrightarrow^* \boldsymbol{M_3}$ ならば，ある $\boldsymbol{M_4}$ が存在し，$\boldsymbol{M_2} \longrightarrow^* \boldsymbol{M_4}$ かつ $\boldsymbol{M_3} \longrightarrow^* \boldsymbol{M_4}$ が成り立つ．

定義 1 (正規形) 項 $\boldsymbol{M}$ がこれ以上簡約できない，すなわち，$\boldsymbol{M} \longrightarrow \boldsymbol{N}$ なる項 $\boldsymbol{N}$ が存在しない時，項 $\boldsymbol{M}$ は正規形 *(normal form)* である，という．

定義 2 (正規化可能) 項 $\boldsymbol{M}$ が正規形に簡約できる，すなわち，$\boldsymbol{M} \longrightarrow^* \boldsymbol{N}$ *(ただし* $\boldsymbol{N}$ は正規形*)* であるような $\boldsymbol{N}$ が存在する時，項 $\boldsymbol{M}$ は正規化可能 *(normalizable)* である，または，$\boldsymbol{M}$ は正規形を持つ，という．

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 Coq における簡約の役割

本節冒頭で述べたように，この簡約は Coq における計算の概念を表しており，`Compute` コマンドや `simpl`，`reflexivity` といった計算による証明を行うためのタクティックのベースになっている．

`reflexivity` タクティックは既に述べたように，ふたつの項が $\longleftrightarrow$ 関係にあるかを調べる機能を持っている．

`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と理解することができる．(まず，合流性のおかげで，正規形は高々ひとつに決まる．また，実は必ず正規形が存在することも保証されている．)

`simpl` タクティックは「うまく単純化できるようであれば単純化する」というような挙動だが，この「うまく」をことばで説明するのが案外難しい．実際，Coq のマニュアルには以下のような記述で `simpl` の挙動が説明されている (強調筆者)．

> These tactics apply to any goal. They try to reduce a term to *something still readable* instead of fully normalizing it. They perform a sort of strong normalization with two key differences:
>
> (中略)
>
> In detail, the tactic simpl first applies $\beta\iota$-reduction. Then, it expands transparent constants and tries to reduce further using $\beta\iota$-reduction. But, when no ι rule is applied after unfolding then δ-reductions are not applied. For instance trying to use simpl on `(plus n O)=n` changes nothing.

ι 簡約 (イオタ簡約) とは，match や if による場合分け (R-MatchZ，R-MatchS，R-IfT，R-IfF)，再帰関数の展開 (R-Fix に相当) のことを指している．また，δ 簡約 (デルタ簡約) とは，定義をその定義内容で置き換えることを示している．ので，上の記述はだいたい,「関数定義を展開して計算を進める．途中，**場合分けを通過しなかったら**関数定義展開の直前まで戻す」といっている．

例えば，`plus n O` という項は，まず，`plus`, `n`, `O` それぞれに対して，$\beta\iota$ 簡約を行おうとするが，これらの簡約はできないので，全てこのままである．その次に，定数 `plus` をその定義内容で展開 (δ 簡約) して，さらに $\beta\iota$ 簡約を進める．すると，β 簡約だけで

```
match n with
  O => O
| S n' => S (plus n' O)
end
```

という項まで到達するが，`match` の部分は計算できないので，“when no ι rule is applied” の条件に該当する．そのため，定義の展開のところまでキャンセルされて，結局，`plus n O` に戻ってしまう．

3.5 型付け関係

項 $\boldsymbol{M}$ の型が $\boldsymbol{T}$ であることを $\boldsymbol{M} : \boldsymbol{T}$ と書く．例えば `S O` : `nat` や

```
fun n:nat => match n with O => 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 $\boldsymbol{\Gamma}$ を加えた三項関係 $\boldsymbol{\Gamma} \vdash \boldsymbol{M} : \boldsymbol{T}$ として表す．$\boldsymbol{\Gamma}$ は，$\boldsymbol{x} : \boldsymbol{T}$ 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]とも呼ばれる．例えば，

$$\texttt{x:nat} \vdash \texttt{S x : nat}$$
$$\texttt{n:nat, b:bool} \vdash \texttt{if b then n else S n : nat}$$
$$\texttt{n:nat} \vdash \texttt{fun b:bool => if b then n else S n : bool -> nat}$$

[7]正確には

$$\begin{array}{rcl} \boldsymbol{\Gamma} & ::= & \bullet \\ & | & \boldsymbol{\Gamma},\ \boldsymbol{x} : \boldsymbol{T} \end{array}$$

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．

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*) とも呼ばれる．以下が，型付け規則である．

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

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

$$\frac{}{\boldsymbol{\Gamma} \vdash \texttt{S} : \texttt{nat -> nat}} \quad \text{(T-Succ)}$$

$$\frac{\boldsymbol{\Gamma} \vdash \boldsymbol{M} : \texttt{nat} \qquad \boldsymbol{\Gamma} \vdash \boldsymbol{N_1} : \boldsymbol{T} \qquad \boldsymbol{\Gamma},\ \boldsymbol{x} : \texttt{nat} \vdash \boldsymbol{N_2} : \boldsymbol{T} \qquad (\boldsymbol{x} \notin \mathit{dom}(\boldsymbol{\Gamma}))}{\boldsymbol{\Gamma} \vdash \texttt{match}\ \boldsymbol{M}\ \texttt{with 0 =>}\ \boldsymbol{N_1}\ \texttt{|\ S}\ \boldsymbol{x}\ \texttt{=>}\ \boldsymbol{N_2}\ \texttt{end} : \boldsymbol{T}} \quad \text{(T-Match)}$$

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

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

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

$$\frac{\boldsymbol{\Gamma},\ \boldsymbol{x} : \boldsymbol{S} \vdash \boldsymbol{M} : \boldsymbol{T} \qquad (\boldsymbol{x} \notin \mathit{dom}(\boldsymbol{\Gamma}))}{\boldsymbol{\Gamma} \vdash \texttt{fun}\ \boldsymbol{x} : \boldsymbol{S}\ \texttt{=>}\ \boldsymbol{M} : \boldsymbol{S}\ \texttt{->}\ \boldsymbol{T}} \quad \text{(T-Fun)}$$

$$\frac{\boldsymbol{\Gamma},\ \boldsymbol{x} : \boldsymbol{S}\ \texttt{->}\ \boldsymbol{T},\ \boldsymbol{y} : \boldsymbol{S} \vdash \boldsymbol{M} : \boldsymbol{T} \qquad (\boldsymbol{x} \notin \mathit{dom}(\boldsymbol{\Gamma})) \qquad (\boldsymbol{y} \notin \mathit{dom}(\boldsymbol{\Gamma}))}{\boldsymbol{\Gamma} \vdash \texttt{fix}\ \boldsymbol{x}\texttt{(}\boldsymbol{y} : \boldsymbol{S}\texttt{)} : \boldsymbol{T}\ \texttt{:=}\ \boldsymbol{M} : \boldsymbol{S}\ \texttt{->}\ \boldsymbol{T}} \quad \text{(T-Fix)}$$

$$\frac{\boldsymbol{\Gamma} \vdash \boldsymbol{M} : \boldsymbol{S}\ \texttt{->}\ \boldsymbol{T} \qquad \boldsymbol{\Gamma} \vdash \boldsymbol{N} : \boldsymbol{S}}{\boldsymbol{\Gamma} \vdash \boldsymbol{M}\ \boldsymbol{N} : \boldsymbol{T}} \quad \text{(T-App)}$$

- 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 $\boldsymbol{S}$ `->` $\boldsymbol{T}$ がつくのは，パラメータの型を $\boldsymbol{S}$ として (文脈に追加して)，本体式に $\boldsymbol{T}$ 型がつく時であることがわかる．再帰関数についても，$\boldsymbol{x}$ 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．

  $\mathit{dom}(\boldsymbol{\Gamma})$ は $\boldsymbol{\Gamma}$ 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，`fun x : nat => fun x : bool => x` には型がつかないように思えるが，これも代入による変数の捕捉回避と同じアイデアを使ってパラメータの名前替えを行い，`fun x : nat => fun x0 : bool => x0` と考える[8]ことで回避できる．

[8] `fun x0 : nat => fun x : bool => x0` 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．

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

$$\vdash \texttt{fun n:nat => match n with O => true | S n' => false end : nat -> bool}$$

の導出木は以下のようになる．(以下 $\Gamma = \texttt{n:nat}$ とする．)

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{n} : \texttt{nat}}\ \text{T-Var} \quad \dfrac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \text{T-True} \quad \dfrac{}{\Gamma, \texttt{n'}:\texttt{nat} \vdash \texttt{false} : \texttt{bool}}\ \text{T-False}}{\Gamma \vdash \texttt{match n with O => true | S n' => false end} : \texttt{bool}}\ \text{T-Match}}{\vdash \texttt{fun n:nat => match n with O => true | S n' => false end} : \texttt{nat -> bool}}\ \text{T-Fun}$$

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

$$\vdash M : \texttt{nat->nat->nat}$$

の導出木を書け.

3.6 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

- O true のような，関数ではない値の適用
- if (fun x : nat => ...) then ... else ... 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できる.

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 O, S N，true，false，fun x : T => N，もしくは，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) $\Gamma \vdash M : T$ 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 fun x : T' => M' もしくは fix x(y : S') : T' := M' の形である.
- M_0 が match M_1 with O => M_2 | S x => M_3 end の形 (ただし M_1 は標準形) ならば，M_1 は O または，S M' の形である.
- M_0 が if M_1 then M_2 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 (`O 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(...(O)...)`，`true`，`false`，`fun` $\boldsymbol{x}$ `:` $\boldsymbol{T}$ `=>` $\boldsymbol{N}$，もしくは，`fix` $\boldsymbol{x}$`(`$\boldsymbol{y}$ `:` $\boldsymbol{S}$`)` `:` $\boldsymbol{T}$ `:=` $\boldsymbol{N}$ 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ことが言える．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) $\boldsymbol{M}$ には `fix` が現れず，かつ，$\boldsymbol{\Gamma} \vdash \boldsymbol{M} : \boldsymbol{T}$ とする．

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

Coq では，`fix` を使っていても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，$\boldsymbol{M} \longleftrightarrow \boldsymbol{N}$ 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，$\boldsymbol{M} \longleftrightarrow \boldsymbol{N}$ を判定する問題が決定可能になるのはなぜか説明せよ．

4 単純型付ラムダ計算+ペア

(Lists.v で登場する) ペアで拡張した単純型付ラムダ計算を示す．

構文

$$\begin{array}{lrcl} \text{(types)} & \boldsymbol{S}, \boldsymbol{T} & ::= & \cdots \mid \boldsymbol{S} * \boldsymbol{T} \\ \text{(terms)} & \boldsymbol{M}, \boldsymbol{N} & ::= & \cdots \mid \texttt{(}\boldsymbol{M_1}\texttt{,}\boldsymbol{M_2}\texttt{)} \\ & & \mid & \texttt{match } \boldsymbol{M} \texttt{ with (}\boldsymbol{x}\texttt{, }\boldsymbol{y}\texttt{) => } \boldsymbol{N} \texttt{ end} \end{array}$$

簡約規則

$$\texttt{match (}\boldsymbol{M_1}\texttt{,}\boldsymbol{M_2}\texttt{) with (}\boldsymbol{x}\texttt{, }\boldsymbol{y}\texttt{) => } \boldsymbol{N}[\boldsymbol{x}, \boldsymbol{y}] \texttt{ end} \longrightarrow \boldsymbol{N}[\boldsymbol{M_1}, \boldsymbol{M_2}] \qquad \text{(R-PMatch)}$$

型付け規則

$$\frac{\boldsymbol{\Gamma} \vdash \boldsymbol{M} : \boldsymbol{S} \qquad \boldsymbol{\Gamma} \vdash \boldsymbol{N} : \boldsymbol{T}}{\boldsymbol{\Gamma} \vdash \texttt{(}\boldsymbol{M}\texttt{, }\boldsymbol{N}\texttt{)} : \boldsymbol{S} * \boldsymbol{T}} \qquad \text{(T-Pair)}$$

$$\frac{\boldsymbol{\Gamma} \vdash \boldsymbol{M} : \boldsymbol{T_1} * \boldsymbol{T_2} \qquad \boldsymbol{\Gamma}, \boldsymbol{x} : \boldsymbol{T_1}, \boldsymbol{y} : \boldsymbol{T_2} \vdash \boldsymbol{N} : \boldsymbol{S} \qquad (\boldsymbol{x}, \boldsymbol{y} \notin \mathit{dom}(\boldsymbol{\Gamma}))}{\boldsymbol{\Gamma} \vdash \texttt{match } \boldsymbol{M} \texttt{ with (}\boldsymbol{x}\texttt{, }\boldsymbol{y}\texttt{) => } \boldsymbol{N} \texttt{ end} : \boldsymbol{S}} \qquad \text{(T-PMatch)}$$

5　単純型付ラムダ計算+自然数リスト

構文

$$
\begin{array}{llll}
\text{(types)} & \boldsymbol{S}, \boldsymbol{T} & ::= & \cdots \\
& & \mid & \texttt{natlist} \\
\\
\text{(terms)} & \boldsymbol{M}, \boldsymbol{N} & ::= & \cdots \\
& & \mid & \texttt{nil} \\
& & \mid & \texttt{cons} \\
& & \mid & \texttt{match}\ \boldsymbol{M}\ \texttt{with nil =>}\ \boldsymbol{N_1}\ \texttt{|}\ \texttt{cons}\ \boldsymbol{x}\ \boldsymbol{y}\ \texttt{=>}\ \boldsymbol{N_2}\ \texttt{end}
\end{array}
$$

簡約規則

$$\texttt{match nil with nil =>}\ \boldsymbol{N_1}\ \texttt{|}\ \texttt{cons}\ \boldsymbol{x}\ \boldsymbol{y}\ \texttt{=>}\ \boldsymbol{N_2}\ \texttt{end} \longrightarrow \boldsymbol{N_1} \qquad \text{(R-MatchN)}$$

$$\texttt{match cons}\ \boldsymbol{M_1}\ \boldsymbol{M_2}\ \texttt{with nil =>}\ \boldsymbol{N_1}\ \texttt{|}\ \texttt{cons}\ \boldsymbol{x}\ \boldsymbol{y}\ \texttt{=>}\ \boldsymbol{N_2}[\boldsymbol{x}, \boldsymbol{y}]\ \texttt{end} \longrightarrow \boldsymbol{N_2}[\boldsymbol{M_1}, \boldsymbol{M_2}] \qquad \text{(R-MatchC)}$$

$$\frac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\begin{array}{l}\texttt{match}\ \boldsymbol{M}\ \texttt{with nil =>}\ \boldsymbol{N_1}\ \texttt{|}\ \texttt{cons}\ \boldsymbol{x}\ \boldsymbol{y}\ \texttt{=>}\ \boldsymbol{N_2}\ \texttt{end} \\ \longrightarrow \texttt{match}\ \boldsymbol{M'}\ \texttt{with nil =>}\ \boldsymbol{N_1}\ \texttt{|}\ \texttt{cons}\ \boldsymbol{x}\ \boldsymbol{y}\ \texttt{=>}\ \boldsymbol{N_2}\ \texttt{end}\end{array}} \qquad \text{(RC-MatchL1)}$$

$$\frac{\boldsymbol{N_1} \longrightarrow \boldsymbol{N_1'}}{\begin{array}{l}\texttt{match}\ \boldsymbol{M}\ \texttt{with nil =>}\ \boldsymbol{N_1}\ \texttt{|}\ \texttt{cons}\ \boldsymbol{x}\ \boldsymbol{y}\ \texttt{=>}\ \boldsymbol{N_2}\ \texttt{end} \\ \longrightarrow \texttt{match}\ \boldsymbol{M}\ \texttt{with nil =>}\ \boldsymbol{N_1'}\ \texttt{|}\ \texttt{cons}\ \boldsymbol{x}\ \boldsymbol{y}\ \texttt{=>}\ \boldsymbol{N_2}\ \texttt{end}\end{array}} \qquad \text{(RC-MatchL2)}$$

$$\frac{\boldsymbol{N_2} \longrightarrow \boldsymbol{N_2'}}{\begin{array}{l}\texttt{match}\ \boldsymbol{M}\ \texttt{with nil =>}\ \boldsymbol{N_1}\ \texttt{|}\ \texttt{cons}\ \boldsymbol{x}\ \boldsymbol{y}\ \texttt{=>}\ \boldsymbol{N_2}\ \texttt{end} \\ \longrightarrow \texttt{match}\ \boldsymbol{M}\ \texttt{with nil =>}\ \boldsymbol{N_1}\ \texttt{|}\ \texttt{cons}\ \boldsymbol{x}\ \boldsymbol{y}\ \texttt{=>}\ \boldsymbol{N_2'}\ \texttt{end}\end{array}} \qquad \text{(RC-MatchL3)}$$

型付け規則

$$\boldsymbol{\Gamma} \vdash \texttt{nil} : \texttt{natlist} \qquad \text{(T-Nil)}$$

$$\boldsymbol{\Gamma} \vdash \texttt{cons} : \texttt{nat -> natlist -> natlist} \qquad \text{(T-Cons)}$$

$$\frac{\begin{array}{cc}\boldsymbol{\Gamma} \vdash \boldsymbol{M} : \texttt{natlist} & \boldsymbol{\Gamma} \vdash \boldsymbol{N_1} : \boldsymbol{T} \\ \boldsymbol{\Gamma}, \boldsymbol{x} : \texttt{nat}, \boldsymbol{y} : \texttt{natlist} \vdash \boldsymbol{N_2} : \boldsymbol{T} & (\boldsymbol{x} \neq \boldsymbol{y} \text{ and } \boldsymbol{x}, \boldsymbol{y} \notin \mathit{dom}(\boldsymbol{\Gamma}))\end{array}}{\boldsymbol{\Gamma} \vdash \texttt{match}\ \boldsymbol{M}\ \texttt{with nil =>}\ \boldsymbol{N_1}\ \texttt{|}\ \texttt{cons}\ \boldsymbol{x}\ \boldsymbol{y}\ \texttt{=>}\ \boldsymbol{N_2}\ \texttt{end} : \boldsymbol{T}} \qquad \text{(T-MatchL)}$$