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
cal14@fos.kuis.kyoto-u.ac.jp
igarashi@kuis.kyoto-u.ac.jp

October 21, 2014

Coq で，プログラムの実行がどのようになされるかや，`reflexivity` を使って証明ができる，つまり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型付ラムダ計算を導入する．

プログラムの実行を表すために「式変形 = 計算過程」とする考え方を採用する．式変形として許されるのはどういうものかを規定する方法を理解するのが大切である．

1 自然数の足し算とかけ算

ここではまず導入として S, O で表現される自然数と足し算，かけ算を表現することを考える．このような計算プロセスを表す理論的枠組みを計算体系 (calculus または computational calculus) という．プログラムを表現したものを項 (term) と呼ぶ．

1.1 項の構文

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

変数，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．

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

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

と矢印を使って表す．数学的には，*Terms* 上の二項関係 $\longrightarrow$ を導入することになる．二項 $\boldsymbol{M}, \boldsymbol{N}$ がこの関係で関係付けられていることを $\boldsymbol{M} \longrightarrow \boldsymbol{N}$ 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 $\boldsymbol{M}$ に対しても，$\mathtt{O} + \boldsymbol{M} \longrightarrow \boldsymbol{M}$ という関係が成立する」ことを示す推論規則である．

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

これは $\boldsymbol{M}$ 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・テンプレートになっている．例えば，$\boldsymbol{M} = \mathtt{S\ O}$ とすれば，

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

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

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

が導ける．R-PlusZ は推論規則の名前である．

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．Coq での `plus`, `mult` との類似を見てとってほしい．

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

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

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

これらをの規則を使うと，

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

や

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

といった関係が導けるが，これらの右辺の次の計算ステップとして考えられる

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

や

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

のように，式の一部分に規則をあてはめて計算を進めるような関係を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．

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

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，

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

とすると，この規則は，

$\texttt{0 + S 0} \longrightarrow \texttt{S 0}$ がいえたなら
$\texttt{S(0 + S 0)} \longrightarrow \texttt{S (S 0)}$ 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 $\texttt{S(0 + S 0)} \longrightarrow \texttt{S (S 0)}$ をいってもよいになる．このような，

1. 規則 R-PlusZ より $\texttt{0 + S 0} \longrightarrow \texttt{S 0}$
2. 規則 RC-Succ より $\texttt{S(0 + S 0)} \longrightarrow \texttt{S (S 0)}$

という関係を確認するための推論過程を

$$\dfrac{\dfrac{}{\texttt{0 + S 0} \longrightarrow \texttt{S 0}}\ \text{R-PlusZ}}{\texttt{S (0 + S 0)} \longrightarrow \texttt{S (S 0)}}\ \text{RC-Succ}$$

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になる．

$$\dfrac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\texttt{S (}\boldsymbol{M}\texttt{)} \longrightarrow \texttt{S (}\boldsymbol{M'}\texttt{)}}\ \text{(RC-Succ)}$$

$$\dfrac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\boldsymbol{M}\ \texttt{+}\ \boldsymbol{N} \longrightarrow \boldsymbol{M'}\ \texttt{+}\ \boldsymbol{N}}\ \text{(RC-PlusL)}$$

$$\dfrac{\boldsymbol{N} \longrightarrow \boldsymbol{N'}}{\boldsymbol{M}\ \texttt{+}\ \boldsymbol{N} \longrightarrow \boldsymbol{M}\ \texttt{+}\ \boldsymbol{N'}}\ \text{(RC-PlusR)}$$

$$\dfrac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\boldsymbol{M}\ \texttt{*}\ \boldsymbol{N} \longrightarrow \boldsymbol{M'}\ \texttt{*}\ \boldsymbol{N}}\ \text{(RC-MultL)}$$

$$\dfrac{\boldsymbol{N} \longrightarrow \boldsymbol{N'}}{\boldsymbol{M}\ \texttt{*}\ \boldsymbol{N} \longrightarrow \boldsymbol{M}\ \texttt{*}\ \boldsymbol{N'}}\ \text{(RC-PlusR)}$$

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．

- $\boldsymbol{M_1} =$ `S O + (S O * S O)`
- $\boldsymbol{M_2} =$ `(S O + S O) * (S (S O) + S O)`
- $\boldsymbol{M_3} =$ `S (S (S (S O))) + O`

練習問題 1.2 また，`S (S O) * S O` が簡約されて `S (S O)` になる過程 (全て) を，`S (S O) * S O` $\longrightarrow$ $\boldsymbol{M_1} \longrightarrow \boldsymbol{M_2} \cdots \longrightarrow$ `S (S O)` となる $\boldsymbol{M_i}$ 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，数学では一般的に「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，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．

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 OCaml や 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 OCaml や 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*) ということも) と関数型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) 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 (`Eval compute` による) 実行と (`reflexivity` で証明できる)「等しさ」を正確に把握する．

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*)[1] を以下の構文で与える．

(types) $\boldsymbol{S}, \boldsymbol{T}$ ::=
- `nat`
- `bool`
- $\boldsymbol{S}$ `->` $\boldsymbol{T}$

(terms) $\boldsymbol{M}, \boldsymbol{N}$ ::=
- $\boldsymbol{x}$
- `O`
- `S`
- `match` $\boldsymbol{M}$ `with O =>` $\boldsymbol{N_1}$ `| S` $\boldsymbol{x}$ `=>` $\boldsymbol{N_2}$ `end`
- `true`
- `false`
- `if` $\boldsymbol{M}$ `then` $\boldsymbol{N_1}$ `else` $\boldsymbol{N_2}$
- `fun` $\boldsymbol{x}$ `:` $\boldsymbol{T}$ `=>` $\boldsymbol{M}$
- `fix` $\boldsymbol{x}$`(`$\boldsymbol{y}$ `:` $\boldsymbol{S}$`)` `:` $\boldsymbol{T}$ `:=` $\boldsymbol{M}$
- $\boldsymbol{M_1}$ $\boldsymbol{M_2}$

[1] 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でる．

- 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，関数 (fun)，再帰関数 (fix)，関数適用を考える．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伸ばして読む．また関数適用は左結合する．例えば，

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

  は全体が関数項で

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

  のことである．

- 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．

- 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．

  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
  ```

  のような自明に止まらない関数も (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 $\boldsymbol{M} \longrightarrow \boldsymbol{N}$ 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，先程と同様に

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

で表される．これは $\boldsymbol{x}$ や $\boldsymbol{T}$ 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・テンプレートになっている．この「雛形」を規則と呼ぶことがある．関係を定義するための規則はすぐ後に見るようにもう少し一般的な形をとる．R-Beta はこの「雛形」につけられた名前である．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)
```
$$\longrightarrow$$
```
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 $\boldsymbol{x}$ に代入される．

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

上の規則を使うと，例えば，

```
if true then n else S m ⟶ n
```

が導かれるが，これらの規則だけでは厳密にいうと

```
S (if true then n else S m) ⟶ S n
```

のように式の一部分に規則をあてはめて計算を進めるような関係を導くことができない．このような部分式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．

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

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，

$$\begin{aligned} \boldsymbol{M} &= \texttt{S} \\ \boldsymbol{N} &= \texttt{if true then n else S m} \\ \boldsymbol{N'} &= \texttt{n} \end{aligned}$$

とすると，この規則は，

$$\texttt{if true then n else S m} \longrightarrow \texttt{n} \text{ がいえたなら}$$
$$\texttt{S (if true then n else S m)} \longrightarrow \texttt{S n} \text{ をいってもよい}$$

ということで，前提条件は規則 R-IfT から満たされるので，結局 $\texttt{S (if true then n else S m)} \longrightarrow \texttt{S n}$ がいえることになる．このような，

1. 規則 R-IfT より $\texttt{if true then n else S m} \longrightarrow \texttt{n}$
2. 規則 RC-App2 より $\texttt{S (if true then n else S m)} \longrightarrow \texttt{S n}$

という関係を確認するための推論過程を

$$\dfrac{\dfrac{}{\texttt{if true then n else S m} \longrightarrow \texttt{n}}\ \text{R-IfT}}{\texttt{S (if true then n else S m)} \longrightarrow \texttt{S n}}\ \text{RC-App2}$$

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-App2 は，導出関係にある二項の横に同じ項を並べても簡約関係にある，すなわち関数適用項 $\boldsymbol{M}\ \boldsymbol{N}$ の $\boldsymbol{N}$ を簡約しても全体として簡約関係にある，ということを示しているが，ラムダ計算では同様に項 (とその部分項) の種類だけ，このような規則が用意されており，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

ということを表している．これは，$(2 \times 4) + (3 \times 5)$ という式を $8 + (3 \times 5)$ に計算することもあるし，$(2 \times 4) + 15$ に計算することもあることに対応していると考えられる．

部分項の簡約を許すための規則は以下のようになる．

$$\frac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\boldsymbol{M}\ \boldsymbol{N} \longrightarrow \boldsymbol{M'}\ \boldsymbol{N}} \quad \text{(RC-App1)}$$

$$\frac{\boldsymbol{N} \longrightarrow \boldsymbol{N'}}{\boldsymbol{M}\ \boldsymbol{N} \longrightarrow \boldsymbol{M}\ \boldsymbol{N'}} \quad \text{(RC-App2)}$$

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

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

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

$$\frac{\boldsymbol{M} \longrightarrow \boldsymbol{M'}}{\texttt{if}\ \boldsymbol{M}\ \texttt{then}\ \boldsymbol{N_1}\ \texttt{else}\ \boldsymbol{N_2} \longrightarrow \texttt{if}\ \boldsymbol{M'}\ \texttt{then}\ \boldsymbol{N_1}\ \texttt{else}\ \boldsymbol{N_2}} \quad \text{(RC-If1)}$$

$$\frac{\boldsymbol{N_1} \longrightarrow \boldsymbol{N_1'}}{\texttt{if } \boldsymbol{M} \texttt{ then } \boldsymbol{N_1} \texttt{ else } \boldsymbol{N_2} \longrightarrow \texttt{if } \boldsymbol{M} \texttt{ then } \boldsymbol{N_1'} \texttt{ else } \boldsymbol{N_2}} \quad \text{(RC-If2)}$$

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

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

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

最後に，複数ステップ (0 ステップ以上) での簡約を表す $\boldsymbol{M} \longrightarrow^* \boldsymbol{N}$ と，簡約を通じて二項が「等しい」ことを示す[2] $\boldsymbol{M} \longleftrightarrow \boldsymbol{N}$ を規則の形で定義する．

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

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

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

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

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

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

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

$\boldsymbol{M} \longleftrightarrow \boldsymbol{N}$ は直感的には，$\boldsymbol{M}$ から $\boldsymbol{N}$ 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) で到着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 $\boldsymbol{M} \longleftrightarrow \boldsymbol{N}$ で関係づけられる時である．

練習問題 3.1 項

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

とする．$\boldsymbol{M}$ は以下の 3 つの項

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

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

練習問題 3.2 `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．

[2] β 同値関係とも呼ばれる

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．

3.4 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，**型環境** (*type environment*)[3] とも呼ばれる．例えば，

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

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．

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

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

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

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

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

[3] 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．

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

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

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

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

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

- 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 `->` 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．

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 $\vdash$ `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 Γ = `n:nat` とする．)

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

練習問題 3.3 `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．

3.5 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

- `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 が値である，とは，M が `S(...(O)...)`，`true`，`false`，`fun` $x : T$ `=>` M，もしくは，`fix` $x(y : S) : T$ `:=` M いずれかの形をしていることをいう．

定理 3 (前進性) $\vdash M : T$ ならば，M は何らかの値であるか，$M \longrightarrow M'$ なる M' が存在する．

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，その結果得られた項は値であることがいえる．

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) 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. $M \longrightarrow M_1 \longrightarrow M_2 \longrightarrow \cdots$ なる項の無限列 $M_1, M_2, \ldots$ は存在しない．(強正規化性)