平成 26 年度

筑波大学情報学群情報科学類

卒業研究論文

題目 型付き λ 計算の強正規化定理の形式化

主専攻　　ソフトウェアサイエンス主専攻

著者　　坂口和彦

指導教員　南出靖彦

要　旨

型付き λ 計算の強正規化定理とは，型の付く λ 項はどのように簡約を進めても有限ステップで正規形に到達するという定理である．本研究では，定理証明器 Coq を用いて単純型付き λ 計算と System F の強正規化定理を形式化した．我々の形式化は Tait と Girard の証明に基いており，それぞれの体系について 3 種類ずつの微妙に異る証明を含んでいる．本論文の前半では，強正規化定理の証明の準備として de Bruijn 表現による束縛変数や代入の形式化，代入補題などの代入に関する性質の自動証明法について述べる．後半では，型付き λ 計算の形式化，強正規化定理の形式化について述べる．

目次

図目次

表目次

第 1 章

序論

型付き λ 計算における重要な定理の 1 つとして**強正規化定理** (*strong normalization theorem*) が知られている．強正規化定理とは，「型の付く λ 項は，どのように簡約を進めても有限ステップで正規形に到達する」という定理である．ML や Haskell などの静的型付けの関数型言語では明示的に再帰関数を書かない限りは繰り返しが記述できないことが知られているが，この性質は強正規化定理によって支えられている．また，直観主義論理の無矛盾性は，その論理に対応する型付き λ 計算の体系の強正規化性から導ける．

本研究では，定理証明器 Coq [29] とその拡張 SSReflect [15] を用いて，単純型付き λ 計算と System F の強正規化定理の形式化 [25] を完成させた．本論文の前半 (第一部) では，強正規化定理の証明の準備として束縛変数や代入の形式化，代入補題などの代入に関する性質の自動証明法について述べる．後半 (第二部) では，型付き λ 計算の形式化，強正規化定理の形式化について述べる．

第一部　λ 計算の形式化において，束縛位置と変数の対応をどのように表現するか，またその表現に対して代入をどのように定義するかは，その後の証明の複雑さに大きく影響する非常に重要な問題である．変数名を使う通常の λ 項の表現では，被束縛の変数の位置から項の外側に向かって束縛位置を順番に見て，最初に同じ変数名が書いてあった束縛位置と元の変数が対応付いていると考える．この表現における証明は，厳密に記述すると束縛変数の名前の付け替え (α 同値関係での書き換え) が必要であり煩雑になる．そこで，本研究ではより形式化に向いた λ 項の表現方法として知られている **de Bruijn 表現**[*1](*de Bruijn representation*) [9] を用いる．De Bruijn 表現の項に対する代入の定義には大きく分けて 2 つの方法があり，一方は証明に向いた定義，もう一方は実装に向いた (証明に向かない) 定義とされている [21]．この 2 つの代入はどちらも単一の変数への代入の形をしているが，強正規化定理の証明では複数の変数への代入 $t[x_1, \ldots, x_n := u_1, \ldots, u_n]$ が必要になる．我々は後者の定義を並列代入に拡張したものを代入の定義として採用しており，並列代入に拡張した場合には後者の定義の方が証明に適していることを明らかにした．

変数名を使う通常の λ 項の表現における**代入補題** (*substitution lemma*) とは，$x \neq y \wedge x \notin \mathrm{FV}(t_2)$ のときに $t[x := t_1][y := t_2] = t[y := t_2][x := t_1[y := t_2]]$ が成り立つという，代入の適

[*1] **名無し表現** (*nameless representation*) とも言う．

用順序の入れ替えに関する補題である．De Bruijn 表現の場合にはこれ以外にも項についての操作に関する同値性を多数示す必要があり，我々はそれらの性質に対する自動証明法を考案した．この自動証明法の重要な部分は，Coq の組み込みタクティクとして実装された等式のソルバ congruence タクティク [29, Section 8.10.5] と量化子を含まない範囲のプレスバーガー算術のソルバ lia (linear integer arithmetic) タクティク [6][29, Section 21.5] によって行われている．ただし，実際には本来それらのタクティクで (原理的に，もしくは現実的に) 解けない問題を解かせるために多くの工夫が必要である．また，一般的には lia で解ける範囲の問題には omega [29, Chapter 20] という別のタクティクを使うことが多いが，我々が解きたい問題に対しては lia の方が時間的性能が良いことを明らかにした．

第二部　第二部では，まず第一部で説明した項や代入の定義方法を拡張して，単純型付き λ 計算と System F を定義する．次に，それらの定義について強正規化定理の証明方法を述べる．

本研究では Tait [28] と Girard [13, 14] の方法に基いて強正規化定理を形式化しているが，その際に使う reducibility という述語の定義にいくつかのやり方があり，それぞれ少しずつ異なる証明が得られることを明らかにした．我々の形式化では，単純型付き λ 計算と System F それぞれについて 3 通りの reducibility の定義を含んでおり，それらの reducibility の定義全てについて強正規化定理の証明を完成させている．

我々の強正規化定理の形式化は，まず Girard の証明 [14, Chapter 6 and 14] を読んで，Coq でなるべく忠実に書き写す作業から始まった．しかし Girard の証明での型付き λ 計算の定義方法と我々の定義は微妙に異なっており，実際には証明に変更を加えない限り上手くいかない．本研究での形式化を通じて，この問題の原因と解決法を明らかにした．

第 2 章

準備

本章では，本論文を読む上で必要な数学的準備を行う．

2.1 自然数

定義 2.1 (自然数) 0 以上の整数を**自然数** (*natural number*) と呼び，その集合 $\{0,1,2,\dots\}$ を $\mathbb{N}$ で書く．

定義 2.2 (後者関数) $\mathrm{S}n$ は自然数 n に 1 を足して得られる数である．この S を**後者関数** (*successor function*) と呼ぶ[*1]．

定義 2.3 (自然数の減算) 自然数の減算 $m \dot{-} n$ を以下のように定義する．本論文中に出現する $m - n$ は原則としてこの $m \dot{-} n$ を意味している．

$$m \dot{-} n = \begin{cases} m - n & \text{if } n \leq m \\ 0 & \text{if } m < n \end{cases}$$

定義 2.4 自然数 m, n のうち，小さい数と大きい数をそれぞれ $\min(m,n)$, $\max(m,n)$ と書く．

$$\min(m,n) = \begin{cases} m & \text{if } m \leq n \\ n & \text{if } n < m \end{cases} \qquad \max(m,n) = \begin{cases} m & \text{if } n \leq m \\ n & \text{if } m < n \end{cases}$$

2.2 オプション

Coq の `option` 型に相当する集合を定義する．

定義 2.5 (オプション) 集合 A と $\{\bot\}$ の直和を A の**オプション**と呼び，$\mathrm{option}\, A$ と書く．A を $\mathrm{option}\, A$ に埋め込む (Coq の `some` に相当する) 関数を明示的に書きたい場合は $\lfloor \cdot \rfloor$ を使う．

[*1] Coq の自然数型は 2 つのコンストラクタ $0 : \mathbb{N}$ と $\mathrm{S} : \mathbb{N} \to \mathbb{N}$ についての帰納的定義となっている．本論文中の後者関数 S は，このコンストラクタ相当のものを表現したい場合のみに近い，そのような意図が無い場合には $+1$ などで表記する．

2.3 列

定義 2.6 (列) 0 個以上の値の並びを**列** (*sequence*) と呼ぶ．特に，集合 A の元からなる列を A の列と呼ぶ．列を表すメタ変数には $\bar{s}$ のようにして上線を付ける．

定義 2.7 $x_0, \ldots, x_{n-1}$ の列は原則として $[x_0, \ldots, x_{n-1}]$ と表記する．ただし，括弧を外しても意味が曖昧にならない場合に限り $x_0, \ldots, x_{n-1}$ と表記しても良い．

定義 2.8 (列の長さ) 列 $\bar{s}$ の長さを $|\bar{s}|$ で表す．

定義 2.9 (列の連結) 列 $\bar{s}$ の先頭に x を付け加えて得られる列を $x, \bar{s}$ と表記する．同様に，列 $\bar{s}$ と列 $\bar{t}$ を連結して得られる列を $\bar{s}, \bar{t}$ と表記する．ただし，そのように表記すると意味が曖昧になる場合に限り，これらを $x :: \bar{s}, \bar{s} +\!\!+ \bar{t}$ と表記する．

定義 2.10 (マップ) 列 $\bar{s} = [x_0, \ldots, x_{n-1}]$ の全ての要素に関数 f をマップして得られる列 $[f(x_0), \ldots, f(x_{n-1})]$ を $[f(x) \mid x \leftarrow \bar{s}]$ と表記する．

定義 2.11 列 $\bar{s}$ の n 番目の要素を $\bar{s}_n$ で表す．ただし，列の先頭は 0 番目とする．また，この記法を使って良いのは明らかに $n < |\bar{s}|$ が成り立つ場合に限る．

定義 2.12 列 $\bar{s}$ の n 番目の要素を表すもう 1 つの方法として，以下の定義を用いる．

$$\mathrm{nth}(d, \bar{s}, n) = \begin{cases} \bar{s}_n & \text{if } n < |\bar{s}| \\ d & \text{if } |\bar{s}| \leq n \end{cases}$$

この nth は $n < |\bar{s}|$ が成り立つとは限らない場合にも使えるようになっている．

定義 2.13 集合 $A_1, \ldots, A_n$ について，$A_1 \times \cdots \times A_n$ の列に組の $i(\leq n)$ 番目を取り出す関数をマップして A_i の列を作る関数を unzip_i で書く．

$$\mathrm{unzip}_i(\bar{s}) = [x_i \mid (x_1, \ldots, x_n) \leftarrow \bar{s}]$$

定義 2.14 列 $\bar{s}$ の先頭 n 要素を $\mathrm{take}(n, \bar{s})$，$n$ 番目以降の要素を $\mathrm{drop}(n, \bar{s})$ と書く．ただし，$|\bar{s}| \leq n$ の場合，$\mathrm{take}(n, \bar{s}) = \bar{s}, \mathrm{drop}(n, \bar{s}) = []$ とする．

定義 2.15 (列の挿入) 列 $\bar{x}$ の n 番目の位置に列 $\bar{y}$ を挿入する操作 insert を，以下のように定義する．

$$\mathrm{insert}(\bar{x}, \bar{y}, d, n) = \mathrm{take}(n, \bar{y}) +\!\!+ [\underbrace{d, \ldots, d}_{n - |\bar{y}|}] +\!\!+ \bar{x} +\!\!+ \mathrm{drop}(n, \bar{y})$$

この定義における d は，$\bar{y}$ の長さが n に満たなかった場合に，n 番目までの要素を埋めるための値である．

補題 2.16 以下が成り立つ．

$$\mathrm{nth}(d, \mathrm{insert}(\bar{x}, \bar{y}, d', n), m) = \begin{cases} \mathrm{nth}(d', \bar{y}, m) & \text{if } m < n \\ \mathrm{nth}(d', \bar{x}, m - n) & \text{if } n \leq m < n + |\bar{x}| \\ \mathrm{nth}(d, \bar{y}, m - |\bar{x}|) & \text{if } n + |\bar{x}| \leq m \end{cases}$$

2.4 環境

定義 2.17 (環境) 集合 A のオプションの列を A の**環境** (*environment*) と呼ぶ．環境を表すメタ変数としては $\Gamma, \Delta, \ldots$ を使う．列 $\bar{s}$ に $\lfloor \cdot \rfloor$ をマップして得られる環境 $[\lfloor x \rfloor \mid x \leftarrow \bar{s}]$ を $\lfloor \bar{s} \rfloor$ と書く．

定義 2.18 環境に限っては Γ_n を以下のようにして再定義する．

$$\Gamma_n = \mathrm{nth}(\bot, \Gamma, n)$$

定義 2.19 (環境の大小) 環境について，以下の順序関係 $\leq$ を導入する．

$$\Gamma \leqq \Delta \overset{\text{def}}{\iff} (\forall n \in \mathbb{N}, x \in A. \Gamma_n = \lfloor x \rfloor \Rightarrow \Delta_n = \lfloor x \rfloor)$$

定義 2.20 (環境の挿入) 環境 Δ の n 番目の位置に環境 Γ を挿入して得られる環境を $\mathrm{insert}(\Gamma, \Delta, n)$ と書く．

$$\mathrm{insert}(\Gamma, \Delta, n) = \mathrm{insert}(\Gamma, \Delta, \bot, n)$$

2.5 二項関係

二項関係 R の反射推移閉包を R^* で書く．

2.6 λ 計算

λ **計算** (λ-*calculi*) [4, 16] は，関数を基本とする計算モデルである．ここでは，最も基本的な体系であるところの**型無し** λ **計算** (*untyped* λ-*calculus*) を例に取り説明する．

変数の可算無限集合を仮定し，変数を表すメタ変数として $x, y, z, \ldots$ を使う．型無し λ 計算の定義は，計算を表す**項** (*term*) の集合と，計算を進めるための項に関する書き換え規則 (これを簡約と呼ぶ) によって与えられる．項の集合を以下のように定義する．

$$t ::= x \mid (t\, t) \mid (\lambda x. t)$$

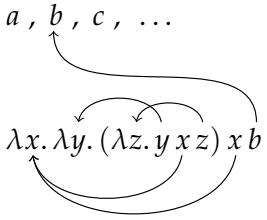
直感的には，$(t_1\, t_2)$ は関数 t_1 に値 t_2 を与えたときの結果であり，$(\lambda x. t)$ は引数を x で取り t を返すような関数である．$(t_1\, t_2)$ の形の項を**適用** (*application*) もしくは**関数適用** (*function application*) と呼び，$(\lambda x. t)$ の形の項を λ **抽象** (λ-*abstraction*) と呼ぶ．これ以降では，原則として以下の規則に従って括弧を省略する．

1. $((t\, u)\, v)$ を $(t\, u\, v)$ と省略する．
2. $(\lambda x. (t\, u))$ を $(\lambda x. t\, u)$ と省略する．
3. $(\lambda x. (\lambda y. t))$ を $(\lambda x. \lambda y. t)$ と省略する．
4. 項の一番外側の括弧を省略する．

更に，以下の規則に従って λ を省略する場合がある．

5. $(\lambda x.\cdots(\lambda y.t)\cdots)$ のように連なっている λ を $(\lambda x\cdots y.t)$ と省略する．

λ 項の書き換え規則を与える前に，どの変数がどの λ に対応しているかを明確にしておく．項中のある変数 x の位置から，上に向かって項を辿って，最初に現れる (つまり，一番内側の) λx は元の変数 x と対応付いていると考える．このような状況のとき，変数 x がその λx の位置で**束縛** (*bind*) されていると言い，対応する λ が存在する変数を**束縛変数** (*bound variable*) と呼ぶ．一方で，対応する λ が存在しない (束縛変数でない) 変数を**自由変数** (*free variable*) と呼ぶ．自由変数を含まない項のことを**閉項** (*closed term*) と呼ぶ．変数の対応関係の例を以下に示す．



t 中の x の自由出現を u で置き換えて得られる項を $t[x := u]$ と書く．λ 項の簡約は，$(\lambda x.t)\,u$ となっている部分項を $t[x := u]$ で置き換える操作 $\to_\beta$ であり，図 2.1 のように定義できる．しかし，代入の定義はこのままだと問題があり，以下のような簡約ができてしまう．

$$\begin{aligned}(\lambda x.\lambda y.x)\,\boldsymbol{y} &\to_\beta (\lambda y.x)[x := \boldsymbol{y}]\\ &= \lambda y.x[x := \boldsymbol{y}]\\ &= \lambda y.\boldsymbol{y}\end{aligned}$$

この例では，簡約前に自由変数だった y が簡約後に束縛変数になっている．代入によって自由変数が束縛変数になってしまう現象を変数の**捕獲** (*capture*) と呼ぶ．このような問題を避けるため，代入は図 2.2[*2]の等式の集まり[*3]によって定義されているものとして，その代わりに

$$\lambda x.t =_\alpha \lambda y.t[x := y] \qquad (y \text{ は } t \text{ 中に出現しない}^{*4})$$

を含む項についての最小の合同関係 $=_\alpha$ で以って項を同一視する．この同値関係 $=_\alpha$ を α **同値関係** (α-equivalence) と呼ぶ．すると，さっきの式変形は

$$\begin{aligned}(\lambda x.\lambda y.x)\,y &\to_\beta (\lambda y.x)[x := y]\\ &=_\alpha (\lambda z.x)[x := y]\\ &= \lambda z.x[x := y]\\ &= \lambda z.y\end{aligned}$$

となり，問題の y は捕獲されていない．

[*2] $\mathrm{FV}(t)$ は，項 t 中の全ての自由変数の集合である．

[*3] $(\lambda y.u)[x := t]$ について，$x = y$ の場合と $x \neq y, y \in \mathrm{FV}(t)$ の場合が含まれておらず，再帰的定義として十分な形にはなっていないことに注意．

[*4] ここでの「出現しない」は自由出現とは限らず，λ の位置に現れる変数も含んでいる．

$$(\lambda x\, t)\, u \to_\beta t[x := u] \qquad \frac{t \to_\beta t'}{t\, u \to_\beta t'\, u} \qquad \frac{u \to_\beta u'}{t\, u \to_\beta t\, u'} \qquad \frac{t \to_\beta t'}{\lambda x.\, t \to_\beta \lambda x.\, t'}$$

図 2.1　型無し λ 計算の β 簡約の定義

$$\begin{aligned} x[x := t] &= t \\ y[x := t] &= y && \text{if } x \neq y \\ (u\, v)[x := t] &= u[x := t]\, v[x := t] \\ (\lambda y.\, u)[x := t] &= \lambda y.\, u[x := t] && \text{if } x \neq y, y \notin \mathrm{FV}(t) \end{aligned}$$

図 2.2　代入の定義

2.7　強正規化性

集合 A と二項関係 $\rightsquigarrow \in A \times A$ について，$x_0 \in A$ が**強正規化可能** (*strongly normalizable*) であるとは，x_0 から始まる任意の $\rightsquigarrow$ での遷移列 $x_0 \rightsquigarrow x_1 \rightsquigarrow \ldots$ が必ず有限長になるということである．このことを $\mathrm{SN}_{\rightsquigarrow}(x_0)$ で表記する．Coq では，標準ライブラリで定義されているアクセシビリティ (Acc) を使うことで強正規化性を表現できる[*5]．Acc の定義を以下に示す．

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
    Acc_intro : (forall y : A, R y x -> Acc A R y) -> Acc A R x.
```

この帰納的定義のコンストラクタと帰納法の原理を論理式で記述すると，以下の SN-Intro と SN-Elim のようになる．

$$\forall x \in A.\, (\forall y \in A.\, x \rightsquigarrow y \Rightarrow \mathrm{SN}_{\rightsquigarrow}(y)) \Rightarrow \mathrm{SN}_{\rightsquigarrow}(x) \qquad \text{(SN-Intro)}$$

$$\forall P \subseteq A.\, (\forall x \in A.\, \mathrm{SN}_{\rightsquigarrow}(x) \wedge (\forall y \in A.\, x \rightsquigarrow y \Rightarrow \mathrm{SN}_{\rightsquigarrow}(y) \wedge P(y)) \Rightarrow P(x)) \Rightarrow \mathrm{SN}_{\rightsquigarrow} \subseteq P \qquad \text{(SN-Elim)}$$

ただし，ここでの帰納法の原理 SN-Elim は Acc の定義から自動的に生成される Acc_ind ではなく，以下の Acc_ind_gen に対応している．

```
Fixpoint Acc_ind_gen
  (A : Type) (R : A -> A -> Prop) (P : A -> Prop)
  (H0 : forall x, Acc R x ->
          (forall y, R y x -> Acc R y) -> (forall y, R y x -> P y) -> P x)
  (x : A) (H : Acc R x) : P x :=
  match H with Acc_intro H1 =>
    H0 x H H1 (fun y H2 => Acc_ind_gen H0 (H1 _ H2))
  end.
```

[*5] ただし，Acc と SN では，関係の向きが逆になっている．

本論文では，原則として SN-Intro と SN-Elim の 2 つの性質のみを公理として使って強正規化性に関する証明を行う．強正規化性の基本的な性質を以下に示す．このうち，補題 2.22, 2.23, 2.24 の証明を付録 B に示す．

補題 2.21 集合 A 上の二項関係 $\prec$ が整礎であることと，$\forall x \in A.\mathrm{SN}_{\succ}(x)$ は同値である[*6]．また，集合 A 上の関係 $\prec$ は，$\{x \in A \mid \mathrm{SN}_{\succ}(x)\}$ 上の整礎関係である．

補題 2.22 以下が成り立つ．

$$\forall x \in A.\mathrm{SN}_{\rightsquigarrow}(x) \Leftrightarrow (\forall y \in A.\, x \rightsquigarrow y \Rightarrow \mathrm{SN}_{\rightsquigarrow}(y))$$

補題 2.23 (SN-Elim′) 以下が成り立つ．

$$\forall P \subseteq A.\,(\forall x \in A.\,(\forall y \in A.\, x \rightsquigarrow y \Rightarrow P(y)) \Rightarrow P(x)) \Rightarrow \mathrm{SN}_{\rightsquigarrow} \subseteq P$$

補題 2.24 任意の二項関係 $\rightsquigarrow_A \subseteq A \times A$, $\rightsquigarrow_B \subseteq B \times B$，関数 $f : A \to B$ について，以下が成り立つ．

$$(\forall x, y \in A.\, x \rightsquigarrow_A y \Rightarrow f(x) \rightsquigarrow_B f(y)) \Rightarrow (\forall x \in A.\mathrm{SN}_{\rightsquigarrow_B}(f(x)) \Rightarrow \mathrm{SN}_{\rightsquigarrow_A}(x))$$

この性質は，整礎関係 $\rightsquigarrow_B$ の関数 f での逆像 $\rightsquigarrow_A$ が整礎関係であることを表している．

補題 2.25 (SN-Elim2) 任意の二項関係 $\rightsquigarrow_A \subseteq A \times A$, $\rightsquigarrow_B \subseteq B \times B$, $P \subseteq A \times B$ について，以下が成り立つ．

$$\begin{aligned}&(\forall x \in A, y \in B.\,\mathrm{SN}_{\rightsquigarrow_A}(x) \wedge \mathrm{SN}_{\rightsquigarrow_B}(y)\\ &\qquad\qquad \wedge (\forall x' \in A.\, x \rightsquigarrow_A x' \Rightarrow \mathrm{SN}_{\rightsquigarrow_A}(x') \wedge P(x', y))\\ &\qquad\qquad \wedge (\forall y' \in B.\, y \rightsquigarrow_B y' \Rightarrow \mathrm{SN}_{\rightsquigarrow_B}(y') \wedge P(x, y')) \Rightarrow P(x, y))\\ &\Rightarrow \mathrm{SN}_{\rightsquigarrow_A} \times \mathrm{SN}_{\rightsquigarrow_B} \subseteq P\end{aligned}$$

定義 2.26 (強正規化性に関する帰納法) 二項関係 $\rightsquigarrow \subseteq A \times A$ と $P \subseteq A$ について，$\forall x.\,\mathrm{SN}_{\rightsquigarrow}(x) \Rightarrow P(x)$ を証明したいとする．この命題は $\mathrm{SN}_{\rightsquigarrow} \subseteq P$ と書いても同じであり，SN-Elim を適用して証明するのであれば，証明すべき命題は以下のようになる．

$$\forall x.\,\mathrm{SN}_{\rightsquigarrow}(x) \wedge (\forall y.\, x \rightsquigarrow y \Rightarrow \mathrm{SN}_{\rightsquigarrow}(y) \wedge P(y)) \Rightarrow P(x)$$

この命題を元の命題 $\forall x.\,\mathrm{SN}_{\rightsquigarrow}(x) \Rightarrow P(x)$ と比較すると，証明中で使える仮定 $\forall y.\, x \rightsquigarrow y \Rightarrow \mathrm{SN}_{\rightsquigarrow}(y) \wedge P(y)$ が増えている以外は何も変化していないことが分かる．

このことを利用して，**強正規化性に関する帰納法**を導入する．$\forall x.\,\mathrm{SN}_{\rightsquigarrow}(x) \Rightarrow P(x)$ という形の命題に強正規化性に関する帰納法を適用して証明するのであれば，元の命題から先頭の $\forall x$ を取り除いた形の命題を，$x \rightsquigarrow y$ についての帰納法の仮定 $\mathrm{SN}_{\rightsquigarrow}(y) \wedge P(y)$ を用いて証明すれば良い．

同様に，$\rightsquigarrow_A \subseteq A \times A$, $\rightsquigarrow_B \subseteq B \times B$, $P \subseteq A \times B$ としたときに，$\forall x \in A, y \in B.\,\mathrm{SN}_{\rightsquigarrow_A}(x) \wedge \mathrm{SN}_{\rightsquigarrow_B}(y) \Rightarrow P(x, y)$ という形の命題についても，SN-Elim2 を根拠として強正規化性に関する帰納法を使える．その場合は，$x \rightsquigarrow_A x'$ についての帰納法の仮定は $\mathrm{SN}_{\rightsquigarrow_A}(x') \wedge P(x', y)$ であり，$y \rightsquigarrow_B y'$ についての帰納法の仮定は $\mathrm{SN}_{\rightsquigarrow_B}(y') \wedge P(x, y')$ である．

[*6] Coq の標準ライブラリにおける整礎関係の定義は，アクセシビリティを使ってこれと同じ形で定義されている．

第 3 章

λ 計算の形式化

本研究の目的は強正規化定理の Coq 上での形式化だが，そのためには λ 項の表現と代入の定義方法について検討する必要がある．本章では，λ 項の表現方法をいくつか紹介した上で，我々が採用した項の表現と代入の定義，それについての代入補題を自動的に証明する方法について説明する．

3.1 λ 項の表現

3.1.1 名前による表現

名前による表現 (*named representation*) とは，第 2.6 節で紹介したような λ 項の表現である．非形式的な証明では，通常この表現が良く使われている．

図 2.2 の代入の定義は単なる等式の集まりであり，再帰的定義として十分な網羅的なものではない．代入の計算を進めるには，必要に応じて α 同値関係を用いて項を変形する必要がある．しかし，以下を図 2.2 の定義に追加することで，再帰的定義として正しい形になる[*1]．

$$(\lambda x.\, u)[x := t] = \lambda x.\, u$$
$$(\lambda y.\, u)[x := t] = \lambda z.\, u[y := z][x := t] \qquad (x \neq y, y \in \mathrm{FV}(t), z \notin \{x\} \cup \mathrm{FV}(t) \cup \mathrm{FV}(u))$$

この形で定義した代入を，**捕獲回避代入** (*capture avoiding substitution*) と呼ぶ．Coq などの証明器で公理を導入せずに代入を定義するとこのような形になる．この表現の上では多くの証明で α 同値関係を明示的に扱う必要があり，証明は場合分けの多い複雑なものになりがちである．

3.1.2 Nominal Logic

Nominal logic [19] は変数に関する理論を扱うために作られた一階の論理である．Nominal logic を使う証明では名前による表現と同じ形の項を扱うが，α 同値な項を同一視できるように論理体系自体に手を加えている．

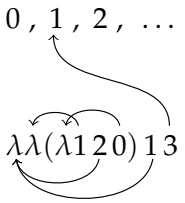
[*1] 実際には，これに加えて x, y, t, u に対する z の選び方を一意に定める必要がある．ただし，条件を満たしている z であればどのように選んでも右辺が α 同値な項になる．

3.1.3 De Bruijn 表現

De Bruijn 表現 [9] では，λ 抽象の位置には変数に関する情報を書かず，変数の側で対応する λ 抽象の番号を持つことで λ 項を表現する．例えば，de Bruijn 表現による型無し λ 計算の項の集合は以下のように定義できる．

$$t ::= x \in \mathbb{N} \mid (t\,t) \mid (\lambda\, t)$$

De Bruijn 表現での束縛と被束縛の対応付けは，被束縛の側から書いてある番号の分だけ項の外側に向かって束縛位置を数えることで得られる．ただし，これを数えている間に項の一番外側に到達してしまった場合，その変数は自由変数であるものとする．この対応の例を以下に示す．



自由変数とインデックスの間の一対一の対応を仮定すると，de Bruijn 表現の項の集合は名前による表現の項の集合を α 同値関係で割ったものに対応する．本研究では，de Bruijn 表現を用いて項を定義する．代入の定義方法などについては，次節で説明する．

3.1.4 Locally Nameless 表現

Locally nameless 表現 [3] は，束縛変数のみを de Bruijn 表現と同様の番号によって表すような λ 項の表現である．この表現では自由変数と束縛変数を構文的に区別し，これらをそれぞれ名前と番号によって表す．

自由変数と束縛変数を構文的に区別して後者を自然数で表すような項のデータ型を素直に定義すると，その項の集合には「束縛変数を束縛する λ が無い項」が入ってしまう．そこで，locally nameless 表現では束縛変数が実際に束縛変数になっている項を**局所閉項** (*locally closed term*) と呼んで区別する．これによって必要な定義が 1 つ増えるが，その一方で de Bruijn 表現で必要なシフト (第 3.2.1 節) と呼ばれる操作は不要である．局所閉項だけを表すような代数的データ型の定義方法 [7, 5] も存在する[*2]が，Coq などの証明器でその定義を採用する場合には証明で heterogeneous equality[*3]を使う必要がある．

Locally nameless 表現の局所閉項の集合は，名前による表現の項の集合を α 同値関係で割ったものに対応する．

[*2] ただし，この定義方法では自由変数と束縛変数は構文的には区別されておらず，一般的には locally nameless とは呼ばない．

[*3] 左右の型が一致しないような等式のこと．Coq でこれを扱うためには，追加の公理が必要になる．

3.1.5 HOAS (Higher-Order Abstract Syntax)

HOAS [22] は，それによって定義しようとしている言語 (ターゲット言語と呼ぶ) の関数を，ターゲット言語を定義するのに使う言語 (ホスト言語と呼ぶ) が元々持っている関数の機能を使って表現するような言語の定義方法である．HOAS では，型無し λ 計算の項の集合 term は以下の 2 つのコンストラクタを持つデータ型として定義する．

$$\begin{aligned} \mathrm{app} &: \mathrm{term} \to \mathrm{term} \to \mathrm{term} \\ \mathrm{abs} &: (\mathrm{term} \to \mathrm{term}) \to \mathrm{term} \end{aligned}$$

ホスト言語の無名関数を $\boldsymbol{\lambda}$ で書いて区別すると，名前による表現で $\lambda x\, y.\, y\, x$ と書ける λ 項は HOAS では $\mathrm{abs}(\boldsymbol{\lambda} x.\, \mathrm{abs}(\boldsymbol{\lambda} y.\, \mathrm{app}(y, x)))$ と書ける．

しかし，このようなデータ型をそのまま Coq 上で定義することはできない．

```
Inductive term : Set := app of term & term | abs of term -> term.
```

```
Error: Non strictly positive occurrence of "term" in
 "(term -> term) -> term".
```

なぜこのような制限があるかというと，もしこのようなデータ型の定義とそれに関する帰納法を許してしまうと，以下のようにして正規形を持たない項 (無限ループ) を構成できてしまうからである．Coq は，全ての式が正規形を持つことを前提として作られている[*4]．

$$\begin{aligned} f(t) &= \begin{cases} t_1 & \text{if } t = \mathrm{app}(t_1, t_2) \\ g(\mathrm{abs}(g)) & \text{if } t = \mathrm{abs}(g) \end{cases} \\ \mathrm{loop} &= f(\mathrm{abs}(f)) \end{aligned}$$

3.1.6 PHOAS (Parametric Higher-Order Abstract Syntax)

第 3.1.5 節で説明したように，Coq では $\mathrm{abs} : (\mathrm{term} \to \mathrm{term}) \to \mathrm{term}$ の形で λ 抽象を表現するような HOAS のデータ型を定義することができない．この制限を回避するために考案されたのが PHOAS [8] である．

PHOAS では，変数の集合が $\mathbb{V}$ であるような項の集合 $\mathrm{term}_{\mathbb{V}}$ を以下のコンストラクタで定義する．

$$\begin{aligned} \mathrm{var}_{\mathbb{V}} &: \mathbb{V} \to \mathrm{term}_{\mathbb{V}} \\ \mathrm{app}_{\mathbb{V}} &: \mathrm{term}_{\mathbb{V}} \to \mathrm{term}_{\mathbb{V}} \to \mathrm{term}_{\mathbb{V}} \\ \mathrm{abs}_{\mathbb{V}} &: (\mathbb{V} \to \mathrm{term}_{\mathbb{V}}) \to \mathrm{term}_{\mathbb{V}} \end{aligned}$$

HOAS では abs の引数として取る関数の型が $\mathrm{term} \to \mathrm{term}$ であるために Coq で定義できない形になっていたが，ここでは $\mathbb{V} \to \mathrm{term}_{\mathbb{V}}$ となっているので問題無く定義できる．

[*4] 同様の制限は，理論的背景として型付き λ 計算を持つ他の定理証明器にも見られる．例えば，Agda2 では `--no-positivity-check` オプションを付けない限りは上で示したようなデータ型を定義することができない．

3.2 De Bruijn 表現による λ 計算の再定義

本節では，本研究で形式化に用いる de Bruijn 表現について，より詳細な事項を説明する．項の集合の定義は第 3.1.3 節で示した通りなので，あとは適切な代入の定義方法を考えれば良い．

3.2.1 単一の変数への代入

De Bruijn 表現の λ 項に対する代入の定義には，以下のような理由から少し工夫が必要である．

- 名前による λ 項の表現では「x の自由出現」の位置にはそのまま x と書いてあるが，de Bruijn 表現では u 中の x の自由出現は x にその変数から項の一番外側までにある λ の数を足した数になる．
- t に自由出現する変数は代入 $u[x := t]$ によって移った先でも自由変数である必要がある．また，t, u それぞれの中での同じ自由変数は代入によって移った先でも同じ自由変数である必要がある[*5]．
- 項 $(\lambda\, u)\, t$ を 1 回簡約すると $u[0 := t]$ になるように代入を定義したい．

この 3 つのことに注意すると，de Bruijn 表現の λ 項に対して通常良く使われる代入は，図 3.2 のように定義できる[*6]．この定義は少し複雑だが，場合分けのそれぞれの部分を見ると以下のような理由から自然な定義になっていることが分かる:

項が変数 y かつ $x = y$ の場合 代入の位置と変数が一致しているので，代入後の項は t となる．

項が変数 y かつ $y < x$ の場合 代入の位置と変数が一致していないので，代入後の項は元の変数 y となる．

項が変数 y かつ $x < y$ の場合 上のケースからの類推で代入後の項は y となりそうだが，代入によって自由変数 x が左辺の項から消えるので，それより大きい変数からは 1 を引くのが自然である．また，簡約を $(\lambda\, t)\, u \to_\beta t[0 := u]$ の形で定義するためには，t 中の 1 以上の自由変数は代入の結果では 1 減らさなければならないが，その点とも整合性が取れている．

項が $u\, v$ の場合 代入をそのまま u と v に分配すれば良い．

項が $\lambda\, u$ の場合 u 中の y の自由出現は $y = 0$ の場合は $\lambda\, u$ の最初の λ に対応する束縛変数であり，$0 < y$ の場合は $\lambda\, u$ 中の $y - 1$ の自由出現である．逆に外側から見ると $\lambda\, u$ 中の y の自由出現は u 中の $y + 1$ の自由出現となっているので，代入の位置は 1 増やす必要がある．

一方で，右辺の項は t ではなく $t \uparrow^1$ となっているが，この $\uparrow^1$ は図 3.1 で定義される

[*5] ただし，t 中の自由変数 y と u 中の自由変数 y を「同じ自由変数」として扱うとは限らない．

[*6] 図 3.3 で定義する別の代入と区別するため，表記を $u\{x := t\}$ の形に変えている．

$$
\begin{aligned}
x \uparrow_c^d &= \begin{cases} x + d & \text{if } c \leq x \\ x & \text{if } x < c \end{cases} \\
(t\, u) \uparrow_c^d &= t \uparrow_c^d \; u \uparrow_c^d \\
(\lambda\, t) \uparrow_c^d &= \lambda\, t \uparrow_{c+1}^d \\
t \uparrow^d &= t \uparrow_0^d
\end{aligned}
$$

図 3.1　型無し λ 計算のシフトの定義

$$
\begin{aligned}
y\{x := t\} &= \begin{cases} y - 1 & \text{if } x < y \\ t & \text{if } x = y \\ y & \text{if } y < x \end{cases} \\
(u\, v)\{x := t\} &= (u\{x := t\})\, (v\{x := t\}) \\
(\lambda\, u)\{x := t\} &= \lambda\, (u\{x + 1 := t \uparrow^1\})
\end{aligned}
$$

図 3.2　型無し λ 計算の代入の定義 (1)

$$
\begin{aligned}
y[x := t] &= \begin{cases} y - 1 & \text{if } x < y \\ t \uparrow^x & \text{if } x = y \\ y & \text{if } y < x \end{cases} \\
(u\, v)[x := t] &= (u[x := t])\, (v[x := t]) \\
(\lambda\, u)[x := t] &= \lambda\, (u[x + 1 := t])
\end{aligned}
$$

図 3.3　型無し λ 計算の代入の定義 (2)

シフト (もしくはリフト) という操作であり，$t \uparrow^d$ は t の全ての自由変数に d を足して得られる項である．ここでもし t の全ての自由変数に 1 を足さずに $(\lambda\, u)\{x := t\} = \lambda\, (u\{x + 1 := \boldsymbol{t}\})$ とすると，以下のようにして t の自由変数 0 が代入後の項で束縛変数になってしまう．

$$
\begin{aligned}
(\lambda\, 1)\{0 := \mathbf{0}\} &= \lambda\, 1\{1 := \mathbf{0}\} \\
&= \lambda\, \mathbf{0}
\end{aligned}
$$

図 3.2 の定義では $\lambda\, u$ の場合に代入する位置に 1 を足して t を 1 つシフトしていたが，$t \underbrace{\uparrow^1 \ldots \uparrow^1}_{n} = t \uparrow^n$ と代入の定義でシフトの量と代入位置が同じだけ増えていることを利用してシフトをまとめたのが図 3.3 の定義である．この定義は Huet [17] による定義と同一であり，我々が形式化に使う代入の定義に近い形になっている．この定義によって得られる代入後の項は図 3.2 の代入とは一般的には異るが，詳しくは第 3.2.3 節で説明する．

3.2.2　並列代入

複数の変数に同時に項を代入できるように定義した代入を**並列代入** (*parallel substitution*) と呼ぶ．我々が形式化したい強正規化定理の証明には並列代入を使う部分があるので，本節では並列代入を定義する．

図 3.2 と図 3.3 の代入の定義を並列代入に拡張すると，それぞれ図 3.4 と図 3.5 の定義が得られる．この定義における $\bar{t} \uparrow^d$ は $[t \uparrow^d |\; t \leftarrow \bar{t}]$ の略記である．$u\{x := \bar{t}\}$ と $u[x := \bar{t}]$ はどちらも直感的には u に自由出現する変数 $x, \ldots, x + |\bar{t}| - 1$ に項 $\bar{t}_0, \ldots, \bar{t}_{|\bar{t}|-1}$ を代入して得られ

$$y\{x := \bar{t}\} = \begin{cases} y - |\bar{t}| & \text{if } x + |\bar{t}| \leq y \\ \bar{t}_{y-x} & \text{if } x \leq y < x + |\bar{t}| \\ y & \text{if } y < x \end{cases}$$

$$(u\,v)\{x := \bar{t}\} = (u\{x := \bar{t}\})\,(v\{x := \bar{t}\})$$

$$(\lambda\,u)\{x := \bar{t}\} = \lambda\,(u\{x+1 := \bar{t}\uparrow^1\})$$

図 3.4　型無し λ 計算の並列代入の定義 (1)

$$y[x := \bar{t}] = \begin{cases} y - |\bar{t}| & \text{if } x + |\bar{t}| \leq y \\ \bar{t}_{y-x}\uparrow^x & \text{if } x \leq y < x + |\bar{t}| \\ y & \text{if } y < x \end{cases}$$

$$(u\,v)[x := \bar{t}] = (u[x := \bar{t}])\,(v[x := \bar{t}])$$

$$(\lambda\,u)[x := \bar{t}] = \lambda\,(u[x+1 := \bar{t}])$$

図 3.5　型無し λ 計算の並列代入の定義 (2)

る項を意味する．これらの定義の今までの定義と大きく違う点は，項が変数 y の場合である．このうち，$x \leq y < x + |\bar{t}|$ の場合は変数が今代入しようとしてる範囲内に入っている場合であり，その変数 y を置き換える項は $\bar{t}_{y-x}$ (もしくは $\bar{t}_{y-x}\uparrow^x$) となる．残りの場合は $y < x$ と $x + |\bar{t}| \leq y$ に分けられるが，このうち後者の結果は $y - 1$ ではなく $y - |\bar{t}|$ となっている．これは，代入の範囲が 1 つの変数ではなく $\bar{t}$ の長さ分の変数になっているからである．

3.2.3　代入の定義の比較

ここまでで 4 通りの代入の定義を示したので，それらの間の関係，違いを明らかにしておく．図 3.2 と図 3.3 で定義される代入がそれぞれ図 3.4 と図 3.5 で定義される代入の $\bar{t}$ の長さを 1 に制限した特別な場合になっていることは明らかだが，その一方で 2 つの並列代入の定義は異る意味を持っている．この違いを捉えるためには，代入の前と後で自由変数の並びがどのように変化しているかを考えれば良い．$u\{x := \bar{t}\}$ も $u[x := \bar{t}]$ も代入の後では u 中の $n + |\bar{t}|$ 以上の自由変数から $|\bar{t}|$ が引かれているという点では同じだが，前者では $\bar{t}$ の自由変数は元々あったのと同じ位置に移り，後者では代入位置 x の分だけ上にずれている．この自由変数の並びの変化を表したのが図 3.7 と図 3.8 である．どちらの図も縦方向の線が左から順に u，代入後の項，$\bar{t}$ の自由変数の並びを表しており，それらの線上の同じ色が付いている部分が代入の前にどの位置にあった自由変数が代入によってどこに移るかを示している．これと同様の図がシフトについても書ける．図 3.1 の定義をもう一度見ると，$t\uparrow^d$ の定義は $t\uparrow_0^d$ となっており，$t\uparrow_c^d$ は t に関して帰納的に定義されている．$t\uparrow_c^d$ は t 中の c 以上の全ての自由変数に d を足して得られる項なので，これについての自由変数の並びの変化は図 3.6 のようになる．

2 つの並列代入の定義について，

$$u[x := \bar{t}] = u\{x := \bar{t}\uparrow^x\}$$

が成り立つ．$\uparrow^0$ は項の集合上の恒等関数なので，$[0 := \bar{t}]$ と $\{0 := \bar{t}\}$ は同じ操作である．図 3.9 の β 簡約の定義で代入が使われるのは 0 への代入だけなので，どちらの代入の定義も β 簡約の定義には同じようにして使える．

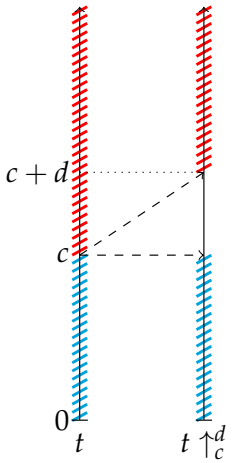


図 3.6　シフトによる自由変数の並びの変化

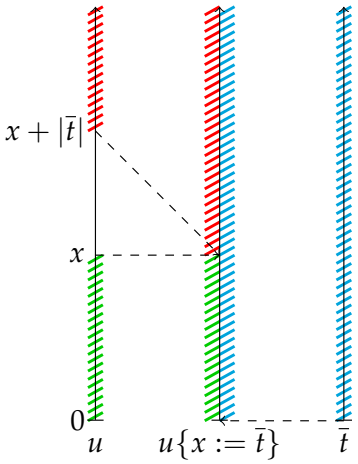


図 3.7　図 3.4 の代入による自由変数の並びの変化

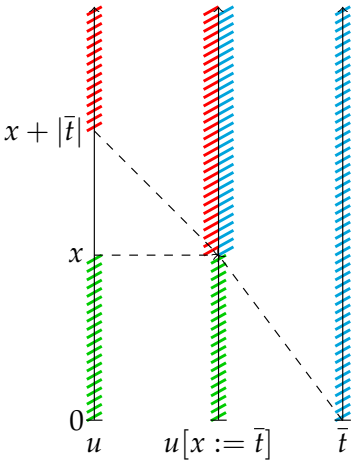


図 3.8　図 3.5 の代入による自由変数の並びの変化

$$(\lambda x.\, t)\, u \to_\beta t\{x := u\} \qquad \frac{t_1 \to_\beta t_2}{t_1\, u \to_\beta t_2\, u} \qquad \frac{u_1 \to_\beta u_2}{t\, u_1 \to_\beta t\, u_2} \qquad \frac{t \to_\beta t'}{\lambda x.\, t \to_\beta \lambda x.\, t'}$$

図 3.9　β 簡約の定義

3.2.4　Coq での定義

我々が実際に Coq での形式化で使っている代入の定義は図 3.5 とほとんど同じだが，Coq で $\bar{t}_{y-x}$ のような式を書くためには $y - x < |\bar{t}|$ の証明を一緒に与える必要がある．そこで，実際には図 3.10 のように nth を使って書き直した定義を使っている．

図 3.5 の定義では $x + |\bar{t}| \le y$ の場合に $y - |\bar{t}|$，$x \le y < x + |\bar{t}|$ の場合には $\bar{t}_{y-x}\uparrow^x$ となっていたが，図 3.10 の定義では以下のようになっている．このことから，実質的にはどちらも同等の定義となっていることが分かる．

- $x + |\bar{t}| \le y$ の場合

$$\begin{aligned}
&\mathrm{nth}(y - x - |\bar{t}|, \bar{t}, y - x)\uparrow^x \\
&= (y - x - |\bar{t}|)\uparrow^x && (|\bar{t}| \le y - x) \\
&= (y - x - |\bar{t}|) + x && (\text{シフトの定義}) \\
&= y + x - x - |\bar{t}| && (x + |\bar{t}| \le y) \\
&= y - |\bar{t}| && (\text{加算と減算の打ち消し})
\end{aligned}$$

- $x \le y < x + |\bar{t}|$ の場合

$$\begin{aligned}
&\mathrm{nth}(y - x - |\bar{t}|, \bar{t}, y - x)\uparrow^x \\
&= \bar{t}_{y-x}\uparrow^x && (y - x < |\bar{t}|)
\end{aligned}$$

Coq での型無し λ 計算の定義を付録 A.2 に示す．

$$
\begin{aligned}
y[x := \overline{t}] &= \begin{cases} \mathrm{nth}(y - x - |\overline{t}|, \overline{t}, y - x) \uparrow^x & \text{if } x \leq y \\ y & \text{if } y < x \end{cases} \\
(u\, v)[x := \overline{t}] &= (u[x := \overline{t}])\, (v[x := \overline{t}]) \\
(\lambda\, u)[x := \overline{t}] &= \lambda\, (u[x + 1 := \overline{t}])
\end{aligned}
$$

図 3.10　型無し λ 計算の並列代入の定義 (3)

第 4 章

代入補題の自動証明

本章では，代入補題だけではなく図 4.1 に示すようなシフトと代入についての多くの性質を Coq 上で自動的に証明する方法を検討する．これらの補題の証明間の依存関係を図 4.2 に示す．これらの補題のうち主要なものはシフトや代入の適用順序を入れ替える (4.3), (4.4), (4.5), (4.7) のような補題である．特に補題 (4.7) は代入と代入の適用順序を入れ替える補題であるため，代入補題に相当する．

4.1 算術式の単純化

De Bruijn 表現の項に対するシフトや代入の定義には自然数についての算術式が多数含まれており，図 4.1 の補題の自動証明でもこの算術式をいかに上手く扱うかが重要となる．そこで，準備として算術式をなるべく単純な形に変形する方法を与える．

我々が扱いたい算術式で最も良く使われる演算は，加算と減算である．そこで，$(x_1 + \cdots + x_m) - (y_{11} + \cdots + y_{1n}) - \cdots - (y_{o1} + \cdots + y_{op})$ のような式の x と y の間で重複している部分を完全に取り除く操作を付録 A.1.2 の `simpl_natarith` タクティクとして実装した[*1]．また，$m \leq n$ についても $m - n = 0$ に書き換えることで同様の単純化を適用できる．

`simpl_natarith` タクティクの主な部分は，減算の両辺の算術式 e_1, e_2 を受け取り，$e_1 - e_2$ とそれを単純化して得られる式が同じであることの証明を返すように実装されている．ただし，実装の都合で打ち消した部分は 0 で置き換えられるようになっているので，それらの部分を以下の補題での書き換えで消去する．

- `addSn` : $\forall m, n \in \mathbb{N}.\, \mathrm{S}\, m + n = \mathrm{S}(m + n)$
- `addnS` : $\forall m, n \in \mathbb{N}.\, m + \mathrm{S}\, n = \mathrm{S}(m + n)$
- `add0n` : $\forall n \in \mathbb{N}.\, 0 + n = n$
- `addn0` : $\forall n \in \mathbb{N}.\, n + 0 = n$
- `sub0n` : $\forall n \in \mathbb{N}.\, 0 - n = 0$
- `subn0` : $\forall n \in \mathbb{N}.\, n - 0 = n$
- `subSS` : $\forall m, n \in \mathbb{N}.\, \mathrm{S}\, m - \mathrm{S}\, n = m - n$
- `min0n` : $\forall n \in \mathbb{N}.\, \min(0, n) = 0$
- `minn0` : $\forall n \in \mathbb{N}.\, \min(n, 0) = 0$
- `max0n` : $\forall n \in \mathbb{N}.\, \max(0, n) = n$
- `maxn0` : $\forall n \in \mathbb{N}.\, \max(n, 0) = n$
- `leq0n` : $\forall n \in \mathbb{N}.\, (0 \leq n) = \mathrm{true}$

[*1] 一方で，我々が採用している減算の定義は第 2.1 節で述べた通り少し特殊であるため，$n - m + m = n$ のような打ち消しができるとは限らない．

$$t \uparrow_n^0 = t \tag{4.1}$$
$$c \le c' \le c + d \Rightarrow t \uparrow_c^d \uparrow_{c'}^{d'} = t \uparrow_c^{d'+d} \tag{4.2}$$
$$c' \le c \Rightarrow t \uparrow_c^d \uparrow_{c'}^{d'} = t \uparrow_{c'}^{d'} \uparrow_{d'+c}^{d} \tag{4.3}$$
$$c \le n \Rightarrow t[n := \overline{u}] \uparrow_c^d = t \uparrow_c^d [d + n := \overline{u}] \tag{4.4}$$
$$n \le c \Rightarrow t[n := \overline{u}] \uparrow_c^d = t \uparrow_{|\overline{u}|+c}^{d} [n := \overline{u} \uparrow_{c-n}^{d}] \tag{4.5}$$
$$c \le n \wedge |\overline{u}| + n \le d + c \Rightarrow t \uparrow_c^d [n := \overline{u}] = t \uparrow_c^{d-|\overline{u}|} \tag{4.6}$$
$$m \le n \Rightarrow t[m := \overline{u}][n := \overline{v}] = t[|\overline{u}| + n := \overline{v}][m := \overline{u}[n - m := \overline{v}]] \tag{4.7}$$
$$t[|\overline{v}| + n := \overline{u}][n := \overline{v}] = t[n := \overline{v} +\!\!+ \overline{u}] \tag{4.8}$$
$$t[n := []] = t \tag{4.9}$$

図 4.1　型無し λ 計算のシフトと代入の性質

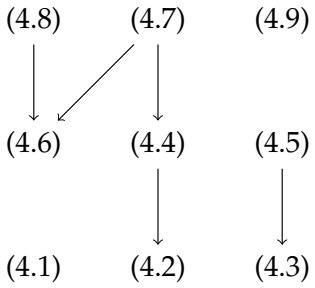


図 4.2　図 4.1 の補題の証明間の依存関係

SSReflect の拡張された `rewrite` タクティクでは，書き換えに使える命題の組を使った書き換えが可能である．つまり，上で列挙した補題について

```
Definition natE :=
  (addSn, addnS, add0n, addn0, sub0n, subn0, subSS,
   min0n, minn0, max0n, maxn0, leq0n).
```

のような定義を書いた上で `rewrite ?natE` と書くことで，これらの補題の左辺から右辺への書き換えを可能な限り繰り返すことができる．このタクティクによる書き換えの例を以下に示す．

$$\begin{aligned}
& a + \mathrm{S}\,0 - \mathrm{S}\,b \\
&= \mathrm{S}(a + 0) - \mathrm{S}\,b && \text{(補題 addnS)} \\
&= \mathrm{S}\,a - \mathrm{S}\,b && \text{(補題 addn0)} \\
&= a - b && \text{(補題 subSS)}
\end{aligned}$$

4.2 仮定の除去

図 4.1 の補題の多くは仮定に数の大小に関する条件を含んでいるが，それらの条件は全て

$$\begin{aligned} &\forall m.\, n \leq m \Rightarrow P[n, m] \\ \Leftrightarrow\ &\forall m'.\, P[n, n + m'] \end{aligned}$$

という性質を用いて除去できる．例えば，補題 (4.5) はこの性質を使うことで

$$t[n := \overline{u}] \uparrow^d_{n+c} = t \uparrow^d_{|\overline{u}|+(n+c)} [n := \overline{u} \uparrow^d_c] \tag{4.10}$$

という形に変形できる．最初からこのような形で命題を書かないのは，Coq の `rewrite` タクティクによる書き換えでこれらの補題を使って証明するときに，証明を簡潔に済ませるためである．例として

$$t \to_\beta t' \Rightarrow t \uparrow^d_c \to_\beta t' \uparrow^d_c$$

の証明を考える．この命題を $t \to_\beta t'$ の導出に関する帰納法で示すのであれば，$t \to_\beta t'$ が $(\lambda\, t_1)\, t_2 \to_\beta t_1[0 := t_2]$ の形の場合については $((\lambda\, t_1 \uparrow^d_{c+1})\, t_2 \uparrow^d_c) \to_\beta t_1[0 := t_2] \uparrow^d_c$ を証明する必要がある．簡約の規則の形にあてはめると，左辺の $((\lambda\, t_1 \uparrow^d_{c+1})\, t_2 \uparrow^d_c)$ から 1 回簡約して得られる項は $t_1 \uparrow^d_{c+1} [0 := t_2 \uparrow^d_c]$ なので，右辺の代入とシフトの適用順序を補題 (4.5) による書き換えで入れ替えたい．ここで，もし条件の不等号を除去した形の命題を使ってしまうと，書き換えの前に $t_1[0 := t_2] \uparrow^d_c$ を $t_1[0 := t_2] \uparrow^d_{0+c}$ の形に直して命題 (4.10) の左辺の形に一致させる必要がある．補題 `add0n` : $\forall n \in \mathbb{N}.\, 0 + n = n$ を使ってこの変形をすると，`rewrite -{3}(add0n c)` のようにして「ゴールに出現する 3 番目の c を `add0n` によって書き換える」と明示することになる．一方で，補題 (4.5) をそのまま使うと $0 \leq c$ というサブゴールは増えるものの，準備無しに $((\lambda\, t_1 \uparrow^d_{c+1})\, t_2 \uparrow^d_c) \to_\beta t_1 \uparrow^d_{c+1} [0 := t_2 \uparrow^d_{c-0}]$ に書き換えられるので，あとは -0 を消して簡約の規則を適用すると証明が完成する．この 2 つの方法での証明の断片は，Coq で書くとそれぞれ以下のようになる．

- `rewrite -{3}(add0n c) subst_shift_distr' /= add1n add0n; auto.`
- `rewrite subst_shift_distr //= add1n subn0; auto.`

この 2 つの証明は後者の方が明らかに簡潔である．また，`rewrite -{3}(add0n c)` のようにして書き換えの位置を出現の番号で指定するのは，証明のメンテナンス性が損なわれる原因となりやすく，証明を書くときに書き換えたい項の出現回数を数える手間も増えるので，その意味でも後者の方が優れていると言える．

図 4.1 の補題の証明は，まず上述の方法で仮定を除去することから始まる．仮定の除去には，付録 A.1.3 の `elimleq` タクティクを用いる．`elimleq` タクティクは，ゴールが $n \leq m \Rightarrow P[n, m]$ の形かつ m が変数のとき，以下のように振る舞う．

1. $n \leq m$ を使ってゴール中に含まれる全ての m を $m - n + n$ で置き換える．
2. m を含む仮定を `move:` で結論側に持ってくる．

3. 元々あった m を消去して $m-n$ に m という名前を付ける[*2].
4. ゴール中の $x+y-y$ や $y+x-y$ という形の部分を x で置き換える.
5. 2 番で結論側に持ってきた仮定を，同じ名前で仮定側に戻す.
6. 第 4.1 節の算術式の単純化を適用する.

4.3 帰納法の適用と等式の自動証明

図 4.1 の全ての補題の証明は，まず上述の方法で仮定を除去した上で，項についての構造的帰納法を適用することから始まる．項には 3 つの形があるのでこれによって 3 つのサブゴールが生成されるが，このうち関数適用の場合と λ 抽象の場合は，両辺にあるシフトと代入の定義を展開すると，

- 帰納法の仮定
- addSn ：$\forall m, n \in \mathbb{N}.\, \mathrm{S}\, m + n = \mathrm{S}(m+n)$
- addnS ：$\forall m, n \in \mathbb{N}.\, m + \mathrm{S}\, n = \mathrm{S}(m+n)$

による書き換えだけで証明できる．例として補題 (4.5) の証明の一部を見る.

Proof. (4.5) の仮定を上述の方法で除去して得られる命題 (4.10) を t に関する構造的帰納法で証明する.

1. t が変数 x のとき
   (ここでは省略する．第 4.4 節を参照．)
2. $t = t_1\, t_2$ のとき

$$
\begin{aligned}
& (t_1\, t_2)[n := \overline{u}] \uparrow^d_{n+c} \\
=\ & (t_1[n := \overline{u}] \uparrow^d_{n+c})\, (t_2[n := \overline{u}] \uparrow^d_{n+c}) && \text{(定義の展開)} \\
=\ & (t_1 \uparrow^d_{|\overline{u}|+(n+c)} [n := \overline{u} \uparrow^d_c])\, (t_2[n := \overline{u}] \uparrow^d_{n+c}) && \text{(I.H. of } t_1\text{)} \\
=\ & (t_1 \uparrow^d_{|\overline{u}|+(n+c)} [n := \overline{u} \uparrow^d_c])\, (t_2 \uparrow^d_{|\overline{u}|+(n+c)} [n := \overline{u} \uparrow^d_c]) && \text{(I.H. of } t_2\text{)} \\
=\ & (t_1\, t_2) \uparrow^d_{|\overline{u}|+(n+c)} [n := \overline{u} \uparrow^d_c] && \text{(定義の畳み込み)}
\end{aligned}
$$

3. $t = \lambda\, t_1$ のとき

$$
\begin{aligned}
& (\lambda\, t_1)[n := \overline{u}] \uparrow^d_{n+c} \\
=\ & \lambda\, t_1[\mathrm{S}\, n := \overline{u}] \uparrow^d_{\mathrm{S}(n+c)} && \text{(定義の展開)} \\
=\ & \lambda\, t_1[\mathrm{S}\, n := \overline{u}] \uparrow^d_{\mathrm{S}\, n+c} && \text{(補題 addSn)} \\
=\ & \lambda\, t_1 \uparrow^d_{|\overline{u}|+(\mathrm{S}\, n+c)} [\mathrm{S}\, n := \overline{u} \uparrow^d_c] && \text{(I.H. of } t_1\text{)} \\
=\ & \lambda\, t_1 \uparrow^d_{|\overline{u}|+\mathrm{S}(n+c)} [\mathrm{S}\, n := \overline{u} \uparrow^d_c] && \text{(補題 addSn)}
\end{aligned}
$$

[*2] 1 番の操作によって m が $m-n$ 以外の形で出現しないようになっているため，この操作ができることが保証されている.

$$= \lambda\, t_1 \uparrow^d_{\mathrm{S}(|\overline{u}|+(n+c))} [\mathrm{S}\, n := \overline{u} \uparrow^d_c] \qquad \text{(補題 addnS)}$$
$$= (\lambda\, t_1) \uparrow^d_{|\overline{u}|+(n+c)} [n := \overline{u} \uparrow^d_c] \qquad \text{(定義の畳み込み)} \qquad \square$$

Coq 上で等式での書き換えだけで証明できる命題を自動的に証明する方法として，Coq の組み込みタクティク congruence がある．congruence タクティクの基本は congruence closure アルゴリズム [20] に基く決定手続きなので，必ず解けるのは量化子を含まない範囲の等式の問題だけである．ただし，実際には全称量化を含む論理式であっても多くの場合は証明できるようになっている．そのため，帰納法で等式を証明しようとするときに書き換えに使う補題を全て仮定に追加しておくと，congruence だけで証明できる場合が多い．例えば，自然数の加算と乗算の可換性と結合性も以下のようにして帰納法と reflexivity と congruence だけで証明できる[*3]．

```
Lemma add0n n : 0 + n = n. Proof. reflexivity. Qed.
Lemma addSn n m : n.+1 + m = (n + m).+1. Proof. reflexivity. Qed.
Lemma addn0 n : n + 0 = n. Proof. elim: n; move: add0n addSn; congruence. Qed.
Lemma addnS n m : n + m.+1 = (n + m).+1.
  Proof. elim: n; move: add0n addSn; congruence. Qed.
Lemma addnC n m : n + m = m + n.
  Proof. elim: n; move: add0n addSn addn0 addnS; congruence. Qed.
Lemma addnA a b c : a + b + c = a + (b + c).
  Proof. elim: a; move: add0n addSn; congruence. Qed.
Lemma mul0n n : 0 * n = 0. Proof. reflexivity. Qed.
Lemma mulSn n m : n.+1 * m = m + n * m. Proof. reflexivity. Qed.
Lemma muln0 n : n * 0 = 0.
  Proof. elim n; move: add0n mul0n mulSn; congruence. Qed.
Lemma mulnS n m : n * m.+1 = n + n * m.
  Proof. elim: n; move: add0n addSn addnC addnA mul0n mulSn; congruence. Qed.
Lemma mulnC n m : n * m = m * n.
  Proof. elim: n; move: mul0n mulSn muln0 mulnS; congruence. Qed.
Lemma mulnDl a b c : (a + b) * c = a * c + b * c.
  Proof. elim: a; move: add0n addSn addnA mul0n mulSn; congruence. Qed.
Lemma mulnA a b c : a * b * c = a * (b * c).
  Proof. elim: a; move: mul0n mulSn mulnDl; congruence. Qed.
```

この方法を利用すると，項が関数適用か λ 抽象の場合については，シフトや代入の定義を simpl で展開した上で，addSn と addnS を仮定に追加してから congruence を実行することで証明を終えられる．この一連の操作は以下の congruence' タクティクにまとめられる．

```
Ltac congruence' := simpl; try (move: addSn addnS; congruence).
```

ここまでの証明の流れをもう一度見ると，第 4.2 節の仮定の除去は同値な命題間の書き換えになっているので，仮定の除去をしないまま帰納法を適用したとしても証明はできるはずである．しかし，その場合は congruence タクティクで証明したい部分がそれで解ける範囲の問題にならないので，先に仮定の除去をするのはそのような意味もある．

[*3] congruence タクティクは本来であれば定義の展開や畳み込みを伴う証明はできないが，この例における add0n, addSn, mul0n, mulSn は自然数の加算と乗算の定義に相当する等式であり，これによって定義の展開と畳み込みに相当する書き換えができる．

4.4 項が変数の場合

前節では，項に関する帰納法を適用した後の 3 つの場合のうち，項が関数適用と λ 抽象の場合については自動的に証明できることを説明した．残りは項が変数の場合だが，シフトや代入の定義は項が変数の場合に数の大小で場合分けをしているので，まずはこの場合分けを全て展開する．例えば，補題 (4.5) については

1. $|\bar{t}| + (n+c) \leq x \wedge n \leq x + d \wedge n \leq x$
   $\Rightarrow \mathrm{nth}(x - n - |\bar{t}|, \bar{t}, x - n) \uparrow^n \uparrow^d_{n+c} = \mathrm{nth}(x + d - n - |\bar{t} \uparrow^d_c|, \bar{t} \uparrow^d_c, x + d - n) \uparrow^n$
2. $x < |\bar{t}| + (n+c) \wedge n \leq x \wedge n \leq x$
   $\Rightarrow \mathrm{nth}(x - n - |\bar{t}|, \bar{t}, x - n) \uparrow^n \uparrow^d_{n+c} = \mathrm{nth}(x - n - |\bar{t} \uparrow^d_c|, \bar{t} \uparrow^d_c, x - n) \uparrow^n$
3. $|\bar{t}| + (n+c) \leq x \wedge x + d < n \wedge n \leq x \Rightarrow \mathrm{nth}(x - n - |\bar{t}|, \bar{t}, x - n) \uparrow^n \uparrow^d_{n+c} = x + d$
4. $x < |\bar{t}| + (n+c) \wedge x < n \wedge n \leq x \Rightarrow \mathrm{nth}(x - n - |\bar{t}|, \bar{t}, x - n) \uparrow^n \uparrow^d_{n+c} = x$
5. $|\bar{t}| + (n+c) \leq x \wedge n \leq x + d \wedge n + c \leq x \wedge x < n$
   $\Rightarrow x + d = \mathrm{nth}(x + d - n - |\bar{t} \uparrow^d_c|, \bar{t} \uparrow^d_c, x + d - n) \uparrow^n$
6. $x < |\bar{t}| + (n+c) \wedge n \leq x \wedge n + c \leq x \wedge x < n$
   $\Rightarrow x + d = \mathrm{nth}(x - n - |\bar{t} \uparrow^d_c|, \bar{t} \uparrow^d_c, x - n) \uparrow^n$
7. $|\bar{t}| + (n+c) \leq x \wedge x + d < n \wedge n + c \leq x \wedge x < n \Rightarrow x + d = x + d$
8. $x < |\bar{t}| + (n+c) \wedge x < n \wedge n + c \leq x \wedge x < n \Rightarrow x + d = x$
9. $\bar{t} + (n+c) \leq x \wedge n \leq x + d \wedge x < n + c \wedge x < n$
   $\Rightarrow x = \mathrm{nth}(x + d - n - |\bar{t} \uparrow^d_c|, \bar{t} \uparrow^d_c, x + d - n) \uparrow^n$
10. $x < |\bar{t}| + (n+c) \wedge n \leq x \wedge x < n + c \wedge x < n \Rightarrow x = \mathrm{nth}(x - n - |\bar{t}| \uparrow^d_c, \bar{t} \uparrow^d_c, x - n) \uparrow^n$
11. $|\bar{t}| + (n+c) \leq x \wedge x + d < n \wedge x < n + c \wedge x < n \Rightarrow x = x + d$
12. $x < |\bar{t}| + (n+c) \wedge x < n \wedge x < n + c \wedge x < n \Rightarrow x = x$

の 12 個の場合がある[*4]．これらの多くの場合の証明は自動化が可能である．例えば，7, 12 の結論の等式は左右の形が一致しているため自明であり，3 は仮定の $x + d < n$ と $n \leq x$ が矛盾している．また，ここには含まれていないが補題 (4.8) については

$$x < |\bar{t}| \wedge x < |\bar{t}| \Rightarrow \mathrm{nth}(v - |\bar{t}|, \bar{t}, x) \uparrow^n = \mathrm{nth}(v - (|\bar{t}| + |\bar{u}|), \bar{t}, x) \uparrow^n$$

を示す必要がある．この場合については仮定の $x < |\bar{t}|$ を使って nth の 1 番目の引数が等式の両辺の計算に使われないことを利用して証明する．

上記のような自明なケースを Coq 上で証明するには omega タクティクが便利である．omega は lia と同様に「量化を含まない範囲のプレスバーガー算術の命題のソルバ」であり，以下のようにして使う．

```
Require Import Omega.
```

[*4] $n \leq x \wedge n \leq x$ のように同じ仮定が複数回出現している箇所もあるが，これらは機械的に場合分けをした結果をそのまま書いている．

```
Goal forall (x y z : nat), x <= y -> y < x - z -> False.
Proof. intros x y z; omega. Qed.
```

しかし，単純に元の問題に omega を適用して解こうとすると現実的な時間で解き終わらない場合が存在する．証明の過程で発見した omega で解けない問題は命題に減算を多く含む傾向があり，そのような問題の小さい例としては以下が考えられる．

```
Notation minn x y := (x - (x - y)) (only parsing).

Lemma minnA (x y z : nat) : minn x (minn y z) = minn (minn x y) z.
Proof. omega. Qed.
```

自然数 x, y について $x-(x-y)$ は $\min(x,y)$ と同じ数なので，補題 minnA は min の結合性を示している．この命題には合計で 10 個の減算が使われており，omega で解ける範囲の問題のはずだが，実際に試してみると非常に長い時間がかかり解けない．その一方で，この omega を lia に差し替えると問題無く解けるが，lia についても命題中の減算の数をもう少し増やすと同様の問題が発生する．

```
Require Import Psatz.

Goal forall (a b c d : nat),
  minn (minn a b) (minn c d) = minn (minn d c) (minn b a).
Proof.
  intros a b c d. Time lia. (* Finished transaction in 76. secs (75.312u,0.056s) *)
Qed.
```

そこで，omega の代わりに以下の手順で証明を行う．ただし，3 以降の手順で 1 つでもサブゴールが残る場合には，2 の操作を終えた時点でのゴールの形に留めておく．

1. 第 4.2 節の仮定の除去と第 4.1 節の算術式の単純化を交互に適用
2. ゴールが自然数間の等式でなければ f_equal [*5]を繰り返し適用
   (ただし，ゴールが $\mathrm{nth}(d_1, \bar{t}, n) = \mathrm{nth}(d_2, \bar{t}, n)$ の形の場合には f_equal の代わりに nth_equal : $\forall a, b, \bar{x}, n.\, (|\bar{x}| \leq n \Rightarrow a = b) \Rightarrow \mathrm{nth}(a, \bar{x}, n) = \mathrm{nth}(b, \bar{x}, n)$ を適用)
3. $m-(m-n)$, $m+(n-m)$, $\min(m,n)$, $\max(m,n)$ の形の式についての場合分け
4. 直前の場合分けで増えた仮定の除去
5. ゴールを lia タクティクが扱える形に変形する[*6]
6. lia タクティクの実行

以上の手順を経て残ったサブゴールについては，手で証明を行う．補題 (4.5) であれば残るのは

1. $\mathrm{nth}(c+x, \bar{t}, \bar{t}+c+x) \uparrow^n \uparrow^d_{n+c} = \mathrm{nth}(|\bar{t}|+c+x+d-|\bar{t}\uparrow^d_c|, \bar{t}\uparrow^d_c, |\bar{t}|+c+x+d) \uparrow^n$
2. $\mathrm{nth}(x-|\bar{t}|, \bar{t}, x) \uparrow^n \uparrow^d_{n+c} = \mathrm{nth}(x-|\bar{t}\uparrow^d_c|, \bar{t}\uparrow^d_c, x) \uparrow^n$

[*5] $f\, x_1 \ldots x_n = f\, y_1 \ldots y_n$ の形のゴールを $x_1 = y_1, \ldots, x_n = y_n$ に変形するタクティク．

[*6] SSReflect の ssrnat.v で定義されている演算や関係は omega や lia などのタクティクでそのまま扱うことができないので，Coq の標準ライブラリで定義されている同等の定義で置き換える必要がある．

表 4.1　図 4.1 の補題の Coq 上での行数

補題	命題の行数	証明の行数	合計
(4.1)	1	1	2
(4.2)	2	1	3
(4.3)	2	1	3
(4.4)	2	4	6
(4.5)	4	6	10
(4.6)	3	4	7
(4.7)	4	6	10
(4.8)	2	4	6
(4.9)	1	1	2
合計	21	28	49

の 2 つであり，最終的に得られた Coq での証明は以下の通りである．

```
Lemma subst_shift_distr n d c ts t :
  n <= c ->
  shift d c (substitute n ts t) =
  substitute n (map (shift d (c - n)) ts) (shift d (size ts + c) t).
Proof.
  elimleq; elim: t n; congruence' => v n; elimif.
  - rewrite !nth_default ?size_map /=; elimif_omega.
  - rewrite -shift_shift_distr // nth_map' /=;
      congr shift; apply nth_equal; rewrite size_map; elimif_omega.
Qed.
```

これとほぼ同様の方法で図 4.1 の全ての補題の証明を完成させた．それらの証明の長さは表 4.1 のようになった．ただし，全ての証明は 1 行の長さが 80 文字以内になるように書いている．また，命題の行数は `Lemma` で始まる行から `Proof.` で始まる行の手前までの行数，証明の行数は `Proof.` から `Qed.` までの行数で数えている．

第 5 章

型付き λ 計算

型付き λ 計算 (*typed λ-calculi*) とは，**型** (*type*) によってある種の制限を付け加えられた λ 計算である．型は，に直感的には λ 計算における値の集合を表すものである．例えば，$\lambda x.\, x$ という項は恒等関数を表しているので，型 U に属する値を受け取ると必ず型 U に属する値を返すはずである．型付き λ 計算では型 U から型 V への関数の型を $U \to V$ で表すので，このことを「項 $\lambda x.\, x$ は型 $U \to U$ を持つ」という形で言い表せる．

本論文の本題であるところの強正規化定理は，型付き λ 計算に関する重要な定理の 1 つである．本章では，強正規化定理の証明の準備として，2 つの型付き λ 計算の体系 (単純型付き λ 計算，System F) の定義と，それらの体系の基本的な性質を述べる．

5.1 単純型付き λ 計算

本節では**単純型付き λ 計算** (*simply typed lambda calculus,* $\lambda^{\to}$) を定義する．まず，型と項の定義を以下に示す．

$$U ::= X \mid (U \to U)$$
$$t ::= x \mid (t\, t) \mid (\lambda x : U.\, t)$$

X は型変数であり，$U \to V$ は上述の通り型 U から型 V への関数の型である．項の定義は，λ 抽象で引数の型を明示するようになっている以外は型無し λ 計算と同じである．第 2.6 節で示した規則に加えて，以下の規則に従って型や項の一部を省略する場合がある．

- 型 $(U \to (V \to W))$ を $(U \to V \to W)$ と省略する．
- 項 $(\lambda x_0 : U_0. \cdots (\lambda x_{n-1} : U_{n-1}.\, t) \ldots)$ を $(\lambda (x_0 : U_0) \ldots (x_{n-1} : U_{n-1}).\, t)$ と省略する．

また，証明中の λ 抽象に付く型情報が重要でない箇所においては，$\lambda x.\, t$ のように省略して表記する場合がある．

明確な定義は示さないが，単純型付き λ 計算についても今までと同様にして代入を定義できる．簡約規則は，その代入を使って図 5.1 のように定義できる．

次に，書ける項の形を型によって制限する方法を与える．変数から型への部分関数のうち，定義域が有限集合であるものを**型環境** (*type environment*) と呼び，そのメタ変数としては Γ, Δ

$$(\lambda x : U.\, t)\, u \to_\beta t[x := u]$$

$$\frac{t \to_\beta t'}{t\, u \to_\beta t'\, u} \qquad \frac{u \to_\beta u'}{t\, u \to_\beta t\, u'} \qquad \frac{t \to_\beta t'}{\lambda x : U.\, t \to_\beta \lambda x : U.\, t'}$$

図 5.1　単純型付き λ 計算の簡約規則

$$\frac{\Gamma(x) = U}{\Gamma \vdash x : U} \qquad \frac{\Gamma \vdash t : U \to V \qquad \Gamma \vdash u : U}{\Gamma \vdash t\, u : V} \qquad \frac{x : U, \Gamma \vdash t : V}{\Gamma \vdash (\lambda x : U.\, t) : U \to V}$$

図 5.2　単純型付き λ 計算の型付け規則

などを使う．型環境は，直感的にはどの自由変数がどの型を持つかを表現するためのものである．定義域が $x_0, \ldots, x_{n-1}$ であり，$\Gamma(x_0) = U_0, \ldots, \Gamma(x_{n-1}) = U_{n-1}$ であるような型環境 Γ を $x_0 : U_0, \ldots, x_{n-1} : U_{n-1}$ と表記する．型環境に新しい変数と型の組を追加するために，以下の定義を用いる．

$$(x : U, \Gamma)(y) = \begin{cases} U & \text{if } x = y \\ \Gamma(y) & \text{if } x \neq y \end{cases}$$

単純型付き λ 計算の型付け規則の定義を図 5.2 に示す．ここで定義されている $\Gamma \vdash t : U$ は「型環境 Γ の下で項 t は型 U を持つ」と読む．これらの規則は左から順にそれぞれ項が変数，関数適用，λ 抽象の場合について，どのような条件を満たすと型が付くかを示している．

単純型付き λ 計算での型付けの例を以下に示す．

$$\dfrac{\dfrac{\dfrac{\dfrac{(f : U \to V, x : U, \Gamma)(f) = (U \to V)}{f : U \to V, x : U, \Gamma \vdash f : U \to V} \qquad \dfrac{(f : U \to V, x : U, \Gamma)(x) = U}{f : U \to V, x : U, \Gamma \vdash x : U}}{f : U \to V, x : U, \Gamma \vdash f\, x : V}}{x : U, \Gamma \vdash \lambda f : U \to V.\, f\, x : (U \to V) \to V}}{\Gamma \vdash \lambda (x : U)(f : U \to V).\, f\, x : U \to (U \to V) \to V}$$

5.2　System F

単純型付き λ 計算における $\lambda x.x$ という項は，λ 抽象に付ける型情報次第で $X \to X$ という型も $(X \to X) \to (X \to X)$ という型も持てる．より一般に，この項は任意の型 U について $U \to U$ という型を持てるはずである．このように，「様々な型を持てる」ことを表すような型を**多相型** (*polymorphic type*) と呼ぶ．*System F* ($\lambda 2$) [13, 14] は，多相型付きの λ 計算である．

System F の型と項の定義を以下に示す．

$$U ::= X \mid (U \to U) \mid (\Pi X.\, U)$$
$$t ::= x \mid (t\, t) \mid (\lambda x : U.\, t) \mid (t\, U) \mid (\Lambda X.\, t)$$

型の定義に新しく導入された $\Pi X.\, U$ が多相型を表しており，この型は直感的には「任意の型 V について，型 $U[X := V]$ を持つ値の型」を意味している．例えば，本節の最初に例とし

$$(\lambda x : U.\, t)\, u \rightsquigarrow t[x := u] \qquad (\Lambda X.\, t)\, U \rightsquigarrow t[X := t]$$

$$\frac{t \rightsquigarrow t'}{t\, u \rightsquigarrow t'\, u} \qquad \frac{u \rightsquigarrow u'}{t\, u \rightsquigarrow t\, u'} \qquad \frac{t \rightsquigarrow t'}{\lambda x : U.\, t \rightsquigarrow \lambda x : U.\, t'}$$

$$\frac{t \rightsquigarrow t'}{t\, U \rightsquigarrow t'\, U} \qquad \frac{t \rightsquigarrow t'}{\Lambda X.\, t \rightsquigarrow \Lambda X.\, t'}$$

図 5.3　System F の簡約規則の定義

$$\frac{\Gamma(x) = U}{\Gamma \vdash x : U} \qquad \frac{\Gamma \vdash t : U \to V \qquad \Gamma \vdash u : U}{\Gamma \vdash t\, u : V} \qquad \frac{x : U, \Gamma \vdash t : V}{\Gamma \vdash (\lambda x : U.\, t) : U \to V}$$

$$\frac{\Gamma \vdash t : \Pi X.\, U}{\Gamma \vdash t\, V : U[X := V]} \qquad \frac{\Gamma \vdash t : U \qquad \forall x.\, X \notin \mathrm{FV}(\Gamma(x))}{\Gamma \vdash \Lambda X.\, t : \Pi X.\, U}$$

図 5.4　System F の型付け規則の定義

て示した多相の恒等関数の型は $\Pi X.\, X \to X$ となる．多相型の導入，除去には，項の定義に追加された $\Lambda X.\, t$ と $t\, U$ を用いる．これらの形の項をそれぞれ**型抽象** (*type abstraction, universal abstraction*)，**型適用** (*type application, universal application*) と呼ぶ．

System F の簡約規則を図 5.3 に示す．この定義の注目すべき点は，規則 $(\Lambda X.\, t)\, U \rightsquigarrow t[X := t]$ が追加されているところである．このような規則があるために System F では (項の) 変数だけではなく型変数についての代入を定義する必要があるが，この代入は $\Pi X.\, U$ と $\Lambda X.\, t$ で型変数が束縛され，$\lambda x : U.\, t$ で変数が束縛されることに注意すれば自然に定義できる．System F の型付け規則を図 5.4 に示す．この定義では $t\, V$ と $\Pi X.\, U$ に対する規則が追加されている．$\Gamma \vdash \Lambda X.\, t : \Pi X.\, U$ となるのは $\Gamma \vdash t : U$ の場合だが，それだけではなく X が Γ 中に出現しないことを要求している．これは，項 $\Lambda X.\, t$ の外から型 X を持ち込めないようにするためである．

System F での型付けの例を以下に示す．ただし，この例での型環境 Γ に型変数 X は自由出現しないものとする．

$$\dfrac{\dfrac{\dfrac{\dfrac{(x : X, \Gamma)(x) = X}{x : X, \Gamma \vdash x : X}}{\Gamma \vdash (\lambda x : X.\, x) : X \to X} \qquad \forall y.\, X \notin \mathrm{FV}(\Gamma(y))}{\Gamma \vdash (\Lambda X.\, \lambda x : X.\, x) : \Pi X.\, X \to X} \qquad \dfrac{\dfrac{(x : U, \Gamma)(x) = U}{x : U, \Gamma \vdash x : U}}{\Gamma \vdash (\lambda x : U.\, x) : U \to U}}{\Gamma \vdash (\Lambda X.\, \lambda x : X.\, x)\, (U \to U) : (U \to U) \to (U \to U)}}{\Gamma \vdash (\Lambda X.\, \lambda x : X.\, x)\, (U \to U)\, (\lambda x : U.\, x) : (U \to U)}$$

$$
\begin{aligned}
X \Uparrow_C^D &= \begin{cases} X + D & \text{if } C \leq X \\ X & \text{if } X < C \end{cases} & x \Uparrow_C^D &= x \\
(U \to V) \Uparrow_C^D &= (U \Uparrow_C^D) \to (V \Uparrow_C^D) & (t\,u) \Uparrow_C^D &= (t \Uparrow_C^D)\,(u \Uparrow_C^D) \\
(\Pi\, U) \Uparrow_C^D &= \Pi\,(U \Uparrow_{C+1}^D) & (\lambda : U.\, t) \Uparrow_C^D &= \lambda : (U \Uparrow_C^D).\, (t \Uparrow_C^D) \\
& & (t\,V) \Uparrow_C^D &= (t \Uparrow_C^D)\,(V \Uparrow_C^D) \\
& & (\Lambda\, t) \Uparrow_C^D &= \Lambda\,(t \Uparrow_{C+1}^D)
\end{aligned}
$$

図 5.5　型変数のシフトの定義

$$
\begin{aligned}
Y[X := \overline{U}] &= \begin{cases} Y - |\overline{U}| & \text{if } X + |\overline{U}| \leq Y \\ \overline{U}_{Y-X} \Uparrow^X & \text{if } X \leq Y < X + |\overline{U}| \\ Y & \text{if } Y < X \end{cases} \\
(V \to W)[X := \overline{U}] &= (V[X := \overline{U}]) \to (W[X := \overline{U}]) \\
(\Pi\, V)[X := \overline{U}] &= \Pi\,(V[X + 1 := \overline{U}]) \\
x[X := \overline{U}] &= x \\
(t\,u)[X := \overline{U}] &= (t[X := \overline{U}])\,(u[X := \overline{U}]) \\
(\lambda : U.\, t)[X := \overline{U}] &= \lambda : (U[X := \overline{U}]).\, (t[X := \overline{U}]) \\
(t\,V)[X := \overline{U}] &= (t[X := \overline{U}])\,(V[X := \overline{U}]) \\
(\Lambda\, t)[X := \overline{U}] &= \Lambda\,(t[X + 1 := \overline{U}])
\end{aligned}
$$

図 5.6　De Bruijn 表現での型変数への代入の定義

5.3　De Bruijn 表現での定義

本節では，System F を de Bruijn 表現で再定義する．単純型付き λ 計算の定義はここから一部分を適切に削れば得られるので，省略する．

De Bruijn 表現での System F の型と項の定義を以下に示す．この定義では型変数，変数の集合はどちらも自然数であり，変数束縛の位置からは変数の情報が消えている．

$$
\begin{aligned}
U &::= X \in \mathbb{N} \mid (U \to U) \mid (\Pi\, U) \\
t &::= x \in \mathbb{N} \mid (t\,t) \mid (\lambda : U.\, t) \mid (t\,U) \mid (\Lambda\, t)
\end{aligned}
$$

De Bruijn 表現での System F の型変数のシフト，型の代入，変数のシフト，項の代入，簡約規則，型付け規則の定義をそれぞれ図 5.5，図 5.6，図 5.7，図 5.8，図 5.9，図 5.10 に示す[*1]．ただし，ここでの型環境 Γ は型の環境 (第 2.4 節) であり，Γ_x が自由変数 x の型を表している．

System F の定義では，項の定義に値の束縛 λ と型の束縛 Λ が含まれているため，図 5.8 のように代入を再帰的に定義しようとすると項が変数 y でかつ代入で置き換わる位置

[*1] 単純型付き λ 計算の場合は，型変数のシフトと型の代入は不要．

$$
\begin{aligned}
x \uparrow_c^d &= \begin{cases} x + d & \text{if } c \le x \\ x & \text{if } x < c \end{cases} \\
(t\,u) \uparrow_c^d &= (t \uparrow_c^d)\,(u \uparrow_c^d) \\
(\lambda : U.\, t) \uparrow_c^d &= \lambda : U.\, (t \uparrow_{c+1}^d) \\
(t\,V) \uparrow_c^d &= (t \uparrow_c^d)\,V \\
(\Lambda\, t) \uparrow_c^d &= \Lambda\, (t \uparrow_c^d)
\end{aligned}
$$

図 5.7　変数のシフトの定義

$$
\begin{aligned}
y[X, x := \overline{t}] &= \begin{cases} y - |\overline{U}| & \text{if } x + |\overline{t}| \le y \\ \overline{t}_{y-x} \uparrow^x \Uparrow^X & \text{if } x \le y < x + |\overline{t}| \\ y & \text{if } y < x \end{cases} \\
(u\,v)[X, x := \overline{t}] &= (u[X, x := \overline{t}])\,(v[X, x := \overline{t}]) \\
(\lambda : U.\, u)[X, x := \overline{t}] &= \lambda : U.\, (u[X, x + 1 := \overline{t}]) \\
(u\,V)[X, x := \overline{t}] &= (u[X, x := \overline{t}])\,V \\
(\Lambda\, u)[X, x := \overline{t}] &= \Lambda\, (u[X + 1, x := \overline{t}])
\end{aligned}
$$

図 5.8　De Bruijn 表現での変数への代入の定義

$$
(\lambda : U.\, t)\, u \rightsquigarrow t[0, 0 := u] \qquad (\Lambda\, t)\, U \rightsquigarrow t[0 := U]
$$

$$
\frac{t \rightsquigarrow t'}{t\,u \rightsquigarrow t'\,u} \qquad \frac{u \rightsquigarrow u'}{t\,u \rightsquigarrow t\,u'} \qquad \frac{t \rightsquigarrow t'}{\lambda : U.\, t \rightsquigarrow \lambda : U.\, t'}
$$

$$
\frac{t \rightsquigarrow t'}{t\,U \rightsquigarrow t'\,U} \qquad \frac{t \rightsquigarrow t'}{\Lambda\, t \rightsquigarrow \Lambda\, t'}
$$

図 5.9　De Bruijn 表現での System F の簡約規則の定義

$$
\frac{\Gamma_x = \lfloor U \rfloor}{\Gamma \vdash x : U} \qquad \frac{\Gamma \vdash t : U \to V \qquad \Gamma \vdash u : U}{\Gamma \vdash t\,u : V} \qquad \frac{\lfloor U \rfloor :: \Gamma \vdash t : V}{\Gamma \vdash \lambda : U.\, t : U \to V}
$$

$$
\frac{\Gamma \vdash t : \Pi\, U}{\Gamma \vdash t\,V : U[0 := V]} \qquad \frac{\Gamma \Uparrow^1 \vdash t : U}{\Gamma \vdash \Lambda\, t : \Pi\, U}
$$

図 5.10　De Bruijn 表現での System F の型付け規則の定義

($x \le y < x + |\overline{t}|$) のときに，値の変数をシフトするだけではなく型変数もシフトする必要がある．そのため，代入自体の定義を $u[X, x := \overline{t}]$ という形にして，型変数をいくつシフトするかを表す X と代入位置 (と値の変数をいくつシフトするか) を表す x を両方持たせている[*2]．また，その代入の定義では項中の型変数のシフトが必要であり，図 5.9 の簡約の定義では項中の型変数への代入が必要になる．これらは，それぞれ図 5.5 の右半分と図 5.6 の下半分で定義されている．

型抽象に対する型付け規則は，通常であれば図 5.4 のようにして型変数 X がその外側に出てこないことを明示する形で定義される．一方，de Bruijn 表現の場合は Λ や Π の内側と外側で自由型変数の数が 1 つずれているので，型環境 Γ の全ての型をシフトすることで「X が Γ に出現しない」相当のことを表現している．

[*2] 簡約の定義で使うときには，今まで通りどちらも 0 とすれば良い．また，単純型付き λ 計算についてはこのような定義にする必要は無く，$u[x := \overline{t}]$ の形で定義すれば十分である．

$$
\begin{aligned}
[f[Y,U] \mid U \leftarrow x]_{Y=X} &= x \\
[f[Y,U] \mid U \leftarrow t\,u]_{Y=X} &= [f[Y,U] \mid U \leftarrow t]_{Y=X}\,[f[Y,U] \mid U \leftarrow u]_{Y=X} \\
[f[Y,U] \mid U \leftarrow \lambda : V.\,t]_{Y=X} &= \lambda : f[X,V].\,[f[Y,U] \mid U \leftarrow t]_{Y=X} \\
[f[Y,U] \mid U \leftarrow t\,W]_{Y=X} &= [f[Y,U] \mid U \leftarrow t]_{Y=X}\,f[X,W] \\
[f[Y,U] \mid U \leftarrow \Lambda\,t]_{Y=X} &= \Lambda\,[f[Y,U] \mid U \leftarrow t]_{Y=X+1}
\end{aligned}
$$

図 5.11　項中の型へのマップの定義

$$
\begin{aligned}
\mathcal{T}(\Gamma,x) &= \Gamma_x \\
\mathcal{T}(\Gamma,t\,u) &= \lfloor U \rfloor && \text{if } \mathcal{T}(\Gamma,t) = \lfloor T \to U \rfloor, \mathcal{T}(\Gamma,u) = \lfloor T \rfloor \\
\mathcal{T}(\Gamma,\lambda : U.\,t) &= \lfloor U \to V \rfloor && \text{if } \mathcal{T}(\lfloor U \rfloor :: \Gamma,t) = \lfloor V \rfloor \\
\mathcal{T}(\Gamma,t\,U) &= \lfloor V[0 := U] \rfloor && \text{if } \mathcal{T}(\Gamma,t) = \lfloor \Pi\,V \rfloor \\
\mathcal{T}(\Gamma,\Lambda X.\,t) &= \lfloor \Pi\,U \rfloor && \text{if } \mathcal{T}(\Gamma \Uparrow^1,t) = \lfloor U \rfloor \\
\mathcal{T}(\Gamma,t) &= \bot && \text{otherwise} \\
\Gamma \vdash t : U &\Leftrightarrow (\mathcal{T}(\Gamma,t) = \lfloor U \rfloor)
\end{aligned}
$$

図 5.12　型付け規則の再定義

5.4　Coq での定義

前節で見た型付き λ 計算の de Bruijn 表現での定義をそのまま Coq で書くと，シフトと代入だけで 6 つの再帰関数を書くことになる．しかし図 5.5 と図 5.6 の下半分はどちらも似たような形をしており，実際にこれを図 5.11 のように項中の型にシフトや代入などの操作を適切にマップする関数として括り出せる．この定義を使うと，$t \Uparrow_C^D$, $t[X := \overline{U}]$ はそれぞれ $[U \Uparrow_X^D \mid U \leftarrow t]_{X=C}$, $[V[Y := \overline{U}] \mid V \leftarrow t]_{Y=X}$ と書ける．

我々は Coq だけではなく SSReflect 拡張を使っているが，SSReflect 拡張を使う場合の述語，関係の定義は `Prop` の代わりに `bool` を使うように，つまり決定手続きとして記述することが推奨されている．そこで，型付け関係は図 5.12 のようにして定義する．$\mathcal{T}$ は型環境と項を受け取って型チェックを行い，もし型 U が付く場合には $\lfloor U \rfloor$ を，型が付かない場合には $\bot$ を返す関数である．

型付き λ 計算に関する証明では，項の構造と型付け規則が一対一に対応していることを利用して，規則の下から上に向かって推論を行う場合がある．例えば，$\Gamma \vdash \lambda : V.\,t : U$ を仮定すると，λ 抽象に対する型付け規則の形から，U は何らかの型 W について $V \to W$ であることが分かる．また，このとき $\lfloor V \rfloor :: \Gamma \vdash t : V \to W$ が導ける．図 5.12 の定義について上述の推論を行うため，以下に示す 8 つの補題を証明した．これらの補題を**逆転補題** (*inversion lemma*)

と呼ぶ．

$$\Gamma \vdash x : U \Leftrightarrow \Gamma_x = \lfloor U \rfloor \tag{5.1}$$
$$\Gamma \vdash t_1\, t_2 : U \Leftrightarrow \exists V.\, (\Gamma \vdash t_1 : V \to U) \wedge (\Gamma \vdash t_2 : V) \tag{5.2}$$
$$\Gamma \vdash (\lambda : V.\, t) : U \Leftrightarrow \exists W.\, U = V \to W \wedge (\lfloor V \rfloor :: \Gamma \vdash t : W) \tag{5.3}$$
$$\Gamma \vdash (\lambda : U.\, t) : U \to V \Leftrightarrow \lfloor U \rfloor :: \Gamma \vdash t : V \tag{5.4}$$
$$\Gamma \vdash (\lambda : U.\, t) : U' \to V \Leftrightarrow U = U' \wedge (\lfloor U \rfloor :: \Gamma \vdash t : V) \tag{5.5}$$
$$\Gamma \vdash t\, U : V \Leftrightarrow \exists W.\, V = W[0 := U] \wedge (\Gamma \vdash t : W) \tag{5.6}$$
$$\Gamma \vdash \Lambda\, t : U \Leftrightarrow \exists U'.\, U = \Pi\, U' \wedge (\Gamma \Uparrow^1 \vdash t : U') \tag{5.7}$$
$$\Gamma \vdash \Lambda\, t : \Pi\, U \Leftrightarrow \Gamma \Uparrow^1 \vdash t : U \tag{5.8}$$

代入は，第 3.2.4 節で説明したのと同様の方法で変形して定義する．単純型付き λ 計算と System F の Coq での定義を，それぞれ付録 A.3 と付録 A.4 に示す．

5.5 型保存定理

型付き λ 計算の基本的な性質として，**型保存定理** (*type preservation theorem*) もしくは**主部簡約定理** (*subject reduction theorem*) と呼ばれる性質が知られている．

定理 5.1 (型保存定理) $t \rightsquigarrow t'$ かつ $\Gamma \vdash t : U$ ならば，$\Gamma \vdash t' : U$ が成り立つ．

定理 5.1 と似たような「型を保存する」性質として，以下が知られている．これらの補題は，System F の型保存定理の証明に必要である．

補題 5.2 (弱化) $(\forall x \in \mathrm{dom}(\Gamma).\, \Gamma(x) = \Delta(x))$ かつ $\Gamma \vdash t : U$ ならば，$\Delta \vdash t : U$ が成り立つ．

補題 5.3 (型の代入についての型の保存) $\Gamma \vdash t : U$ ならば，$\Gamma[X := V] \vdash t[X := V] : U[X := V]$ が成り立つ．

補題 5.4 (代入についての型の保存) $x : V, \Gamma \vdash t : U$ かつ $\Gamma \vdash u : V$ ならば，$\Gamma \vdash t[x := u] : U$ が成り立つ．

このような型の保存に関する補題，定理は，一部の強正規化定理の証明でも使われるため，Coq 上でも形式化したい．De Bruijn 表現を使って型保存定理を証明するためには，代入だけではなくシフトについても同様の性質を示す必要がある．最終的に，System F については以下の 6 つの補題を証明した．

$$\Gamma \trianglelefteq \Delta \wedge \Gamma \vdash t : U \Rightarrow \Delta \vdash t : U \tag{5.9}$$
$$\Gamma \vdash t : U \Rightarrow \Gamma \Uparrow_C^D \vdash t \Uparrow_C^D : U \Uparrow_C^D \tag{5.10}$$
$$\Gamma \vdash t : U \Rightarrow \Gamma[X := \overline{V}] \vdash t[X := \overline{V}] : U[X := \overline{V}] \tag{5.11}$$
$$\Gamma \vdash t : U \Rightarrow \mathrm{insert}(\Gamma, \Delta, c) \vdash t \uparrow_c^{|\Delta|} : U \tag{5.12}$$
$$\begin{aligned}&(\forall (u, U) \in \overline{p}.\, \mathrm{drop}(x, \Gamma) \vdash u \Uparrow^X : U)\\ &\quad \wedge \mathrm{insert}(\lfloor \mathrm{unzip}_2(\overline{p}) \rfloor, \Gamma, x) \vdash t : U \Rightarrow \Gamma \vdash u[X, x := \mathrm{unzip}_1(\overline{p})] : U\end{aligned} \tag{5.13}$$

$$t \rightsquigarrow u \wedge \Gamma \vdash t : U \Rightarrow \Gamma \vdash u : U \tag{5.14}$$

単純型付き λ 計算については，上の補題 (5.9) と補題 (5.12) に加えて，以下を証明した．

$$\begin{aligned} &(\forall (u, U) \in \overline{p}.\, \mathrm{drop}(x, \Gamma) \vdash u : U) \\ &\wedge \mathrm{insert}(\lfloor \mathrm{unzip}_2(\overline{p}) \rfloor, \Gamma, x) \vdash t : U \Rightarrow \Gamma \vdash u[x := \mathrm{unzip}_1(\overline{p})] : U \qquad (5.15) \\ &t \to_\beta u \wedge \Gamma \vdash t : U \Rightarrow \Gamma \vdash u : U \qquad (5.16) \end{aligned}$$

第 6 章

単純型付き λ 計算の強正規化定理

強正規化定理 (*strong normalization theorem*) とは以下のような定理である．

定理 6.1 (強正規化定理) 型の付く λ 項は，簡約の二項関係について強正規化可能である．

強正規化定理の証明は，そのままの形では項や型の構造に関する帰納法を適用しても証明できないことが知られている．良く知られた証明方法としては，以下の 2 つがある．

- 型付き λ 計算の項から簡約列の長さの上界を求める関数を定義し，簡約に関してその値が減少することを証明する [12].
- Reducibility と呼ばれる項に関する述語を定義し，reducible ならば強正規化可能であることと，型が付けば reducible であることを分けて証明する．

我々は，後者の方法に基くいくつかの証明を形式化した．本章と次章では，単純型付き λ 計算と System F の強正規化定理の証明と，その形式化について述べる．また，本章と次章で出現する強正規化性は全て $\to_\beta$ や $\rightsquigarrow$ についての強正規化性なので，これ以降では $\mathrm{SN}_{\to_\beta}$ や $\mathrm{SN}_{\rightsquigarrow}$ を単に SN と書く．

6.1 証明

本節では，Tait の方法 [28][14, Chapter 6] に沿って単純型付き λ 計算の強正規化定理を証明する．通常，型付き λ 計算における項というのは，型の付く項だけを指す言葉である．しかし，本節での証明における λ 項は，第 5.1 節で示した項の定義で生成される全ての項 (これを raw term と呼ぶ) を指しているものとする．

まず，reducibility と neutrality を定義する．

定義 6.2 (Reducibility) 型 U について，項の集合 RED_U を以下のように定義する．

$$\mathrm{RED}_X(t) \overset{\mathrm{def}}{\iff} \mathrm{SN}(t)$$
$$\mathrm{RED}_{U\to V}(t) \overset{\mathrm{def}}{\iff} \forall u.\, \mathrm{RED}_U(u) \Rightarrow \mathrm{RED}_V(t\,u)$$

$\mathrm{RED}_U(t)$ のとき，「項 t は型 U について reducible」という．

定義 6.3 (Neutrality) t が変数か関数適用のどちらかであるとき，項 t は *neutral* であるという．

$$\mathrm{neutral}(t) = \begin{cases} \text{false} & \text{if } t = \lambda x : U.\, t' \\ \text{true} & \text{otherwise} \end{cases}$$

補題 6.4 (CR) 以下が成り立つ．

$$\mathrm{RED}_U(t) \Rightarrow \mathrm{SN}(t) \qquad (\mathrm{CR}_1)$$
$$t \to_\beta t' \wedge \mathrm{RED}_U(t) \Rightarrow \mathrm{RED}_U(t') \qquad (\mathrm{CR}_2)$$
$$\mathrm{neutral}(t) \wedge (\forall t'.\, t \to_\beta t' \Rightarrow \mathrm{RED}_U(t')) \Rightarrow \mathrm{RED}_U(t) \qquad (\mathrm{CR}_3)$$

これ以降ではまず CR_2 を証明し，その次に $\mathrm{CR}_{1,3}$ を証明する．

Proof. CR_2 を型 U に関する帰納法で証明する．

- $U = X$ の場合

  定義を展開すると $t \to_\beta t' \wedge \mathrm{SN}(t) \Rightarrow \mathrm{SN}(t')$ となるので，補題 2.22 より明らか．

- $U = V \to W$ の場合

  定義を展開すると

  $$\begin{aligned} & t \to_\beta t' && \text{(仮定 1)} \\ & \wedge (\forall u'.\, \mathrm{RED}_V(u') \Rightarrow \mathrm{RED}_W(t\, u')) && \text{(仮定 2)} \\ \Rightarrow\ & \forall u.\, \mathrm{RED}_V(u) && \text{(仮定 3)} \\ \Rightarrow\ & \mathrm{RED}_W(t'\, u) \end{aligned}$$

  となる．これは以下のように証明できる．

  $$\begin{aligned} & \mathrm{RED}_V(u) && \text{(仮定 3)} \\ \Rightarrow\ & \mathrm{RED}_W(t\, u) && \text{(仮定 2)} \\ \Rightarrow\ & \mathrm{RED}_W(t'\, u) && \text{(仮定 1, I.H. of } W\text{)} \end{aligned}$$

□

Proof. $\mathrm{CR}_{1,3}$ は型 U に関する帰納法で同時に (互いを帰納法の仮定として使える形で) 証明する．CR_1 の証明を以下に示す．

- $U = X$ の場合

  $\mathrm{RED}_X(t)$ は定義を展開すると $\mathrm{SN}(t)$ となるので明らか．

- $U = V \to W$ の場合

  $$\begin{aligned} & \mathrm{RED}_{V \to W}(t) \\ \Rightarrow\ & \forall u.\, \mathrm{RED}_V(u) \Rightarrow \mathrm{RED}_W(t\, u) && \text{(RED を展開)} \\ \Rightarrow\ & \mathrm{RED}_V(x) \Rightarrow \mathrm{RED}_W(t\, x) && \text{(フレッシュな変数 } x \text{ を導入)} \\ \Rightarrow\ & \mathrm{RED}_W(t\, x) && \text{(I.H. of } V\text{, } \mathrm{CR}_3\text{)} \\ \Rightarrow\ & \mathrm{SN}(t\, x) && \text{(I.H. of } W\text{, } \mathrm{CR}_1\text{)} \\ \Rightarrow\ & \mathrm{SN}(t) && \text{(補題 2.24)} \end{aligned}$$

CR_3 の証明を以下に示す．

- $U = X$ の場合

$$
\begin{aligned}
&\mathrm{neutral}(t) \wedge (\forall t'.\, t \to_\beta t' \Rightarrow \mathrm{RED}_X(t')) \\
\Rightarrow\ &\forall t'.\, t \to_\beta t' \Rightarrow \mathrm{RED}_X(t') \\
\Rightarrow\ &\forall t'.\, t \to_\beta t' \Rightarrow \mathrm{SN}(t') && \text{(RED を展開)} \\
\Rightarrow\ &\mathrm{SN}(t) && \text{(補題 2.22)} \\
\Rightarrow\ &\mathrm{RED}_X(t) && \text{(RED を畳み込み)}
\end{aligned}
$$

- $U = V \to W$ の場合

  ここで証明すべき命題は，定義を展開すると

$$
\begin{aligned}
&\mathrm{neutral}(t) && \text{(仮定 1)} \\
&\wedge (\forall t', u'.\, t \to_\beta t' \wedge \mathrm{RED}_V(u') \Rightarrow \mathrm{RED}_W(t'\,u')) && \text{(仮定 2)} \\
\Rightarrow\ &\forall u.\, \mathrm{RED}_V(u) && \text{(仮定 3)} \\
\Rightarrow\ &\mathrm{RED}_W(t\,u)
\end{aligned}
$$

  となる．仮定 3 と V, CR_1 についての帰納法の仮定より，$\mathrm{SN}(u)$ が成り立つ．

  以下の形に命題を変形し，強正規化性に関する帰納法で証明する．

$$
\begin{aligned}
&\forall u.\, \mathrm{SN}(u) \\
\Rightarrow\ &\mathrm{RED}_V(u) && \text{(仮定 3′)} \\
\Rightarrow\ &\mathrm{RED}_W(t\,u)
\end{aligned}
$$

  まず，$\forall v.\, t\,u \to_\beta v \Rightarrow \mathrm{RED}_W(v)$ を $t\,u \to_\beta v$ の証明と v についての場合分けで証明する．

  - $(\lambda x.\, t')\,u \to_\beta t'[x := u]$ の場合 $(t = \lambda x.\, t', v = t'[x := u])$

    仮定 1 の $\mathrm{neutral}(\lambda x.\, t')$ より，矛盾が導ける．

  - $\dfrac{t \to_\beta t'}{t\,u \to_\beta t'\,u}$ の場合 $(v = t'\,u)$

    仮定 2, 3′ より，$\mathrm{RED}_W(t'\,u)$ が成り立つ．

  - $\dfrac{u \to_\beta u'}{t\,u \to_\beta t\,u'}$ の場合 $(v = t\,u')$

    仮定 3′ と CR_2 より，$\mathrm{RED}_V(u')$ が成り立つ．$u \to_\beta u'$ についての帰納法の仮定より，$\mathrm{RED}_W(t\,u')$ が成り立つ．

  これで $\forall v.\, t\,u \to_\beta v \Rightarrow \mathrm{RED}_W(v)$ が示せた．W, CR_3 についての帰納法の仮定より，$\mathrm{RED}_W(t\,u)$ が成り立つ． □

CR_1 は reducible な項が強正規化可能なことを示しているので，あとは型が付く項が reducible であることを示せば良い．その準備として，以下の補題を証明する．

補題 6.5 $\forall v.\, \mathrm{RED}_U(v) \Rightarrow \mathrm{RED}_V(t[x := v])$ ならば，$\mathrm{RED}_{U\to V}(\lambda x : U.\, t)$ が成り立つ．

Proof. $\mathrm{RED}_{U\to V}(\lambda x : U.\, t)$ は，定義を展開すると $\forall u.\, \mathrm{RED}_U(u) \Rightarrow \mathrm{RED}_V((\lambda x : U.\, t)\, u)$ となる．よって，

$$\begin{aligned} &\forall t.\, (\forall v.\, \mathrm{RED}_U(v) \Rightarrow \mathrm{RED}_V(t[x := v])) && \text{(仮定 1)}\\ \Rightarrow\ &\forall u.\, \mathrm{RED}_U(u) && \text{(仮定 2)}\\ &\Rightarrow \mathrm{RED}_V((\lambda x : U.\, t)\, u) \end{aligned}$$

を示せば良い．まず，t と u が強正規化可能であることを示す．

仮定 1, 2 より，$\mathrm{RED}_V(t[x := u])$ が成り立つ．よって，CR_1 より $t[x := u]$ は強正規化可能である．代入は簡約を保存するので，補題 2.24 より t は強正規化可能である．仮定 2 と CR_1 より，u も強正規化可能である．

ここまでで得られた結果を並べた以下の命題を，強正規化性に関する帰納法で証明する．

$$\begin{aligned} &\forall t, u.\, \mathrm{SN}(t) \wedge \mathrm{SN}(u)\\ &\quad\Rightarrow (\forall v.\, \mathrm{RED}_U(v) \Rightarrow \mathrm{RED}_V(t[x := v])) && \text{(仮定 1′)}\\ &\quad\wedge \mathrm{RED}_U(u) && \text{(仮定 2′)}\\ &\quad\Rightarrow \mathrm{RED}_V((\lambda x : U.\, t)\, u) \end{aligned}$$

$(\lambda x : U.\, t)\, u$ は neutral なので，結論部分に CR_3 を適用できる．よって，以下の命題を証明すれば良い．

$$\begin{aligned} &\forall w.\, (\lambda x : U.\, t)\, u \to_\beta w && \text{(仮定 3)}\\ &\quad\Rightarrow \mathrm{RED}_V(w) \end{aligned}$$

仮定 3 の証明と w について場合分けして証明を行う．

- $(\lambda x : U.\, t)\, u \to_\beta t[x := u]$ の場合 $(w = t[x := u])$

  仮定 1′, 2′ より明らか．

- $\dfrac{t \to_\beta t'}{(\lambda x : U.\, t)\, u \to_\beta (\lambda x : U.\, t')\, u}$ の場合 $(w = (\lambda x : U.\, t')\, u)$

  代入は簡約を保存するので，$t[x := v] \to_\beta t'[x := v]$ が成り立つ．よって，仮定 1′ と CR_2 より $\forall v.\, \mathrm{RED}_U(v) \Rightarrow \mathrm{RED}_V(t'[x := v])$ が成り立つ．$t \to_\beta t'$ についての帰納法の仮定と仮定 2′ より，$\mathrm{RED}_V((\lambda x : U.\, t')\, u)$ が成り立つ．

- $\dfrac{u \to_\beta u'}{(\lambda x : U.\, t)\, u \to_\beta (\lambda x : U.\, t)\, u'}$ の場合 $(w = (\lambda x : U.\, t)\, u')$

  仮定 2′ と CR_2 より，$\mathrm{RED}_U(u')$ が成り立つ．$u \to_\beta u'$ についての帰納法の仮定と仮定 1′ より，$\mathrm{RED}_V((\lambda x : U.\, t)\, u')$ が成り立つ． □

上の補題を使って，残りを証明する．

定理 6.6 (Reducibility) 変数 $x_1, \ldots, x_n$，型 $V_1, \ldots, V_n$，項 $u_1, \ldots, u_n$ について，

- $\forall i \leq n. \mathrm{RED}_{V_i}(u_i)$
- $x_1 : V_1, \ldots, x_n : V_n \vdash t : U$

ならば，$\mathrm{RED}_U(t[x_1, \ldots, x_n := u_1, \ldots, u_n])$ が成り立つ．

Proof. これ以降では，$x_1 : V_1, \ldots, x_n : V_n$ を Γ と書く． $\Gamma \vdash t : U$ の証明に関する帰納法で示す．

- $\dfrac{\Gamma(y) = U}{\Gamma \vdash y : U}$ の場合

  適当な $i \leq n$ について $y = x_i$ かつ $U = V_i$ が成り立つ．このとき，$y[x_1, \ldots, x_n := u_1, \ldots, u_n]$ は u_i となるので，$\mathrm{RED}_{V_i}(u_i)$ を示せば良い．これは $\forall i \leq n. \mathrm{RED}_{V_i}(u_i)$ より明らか．

- $\dfrac{\Gamma \vdash t_1 : W \to U \qquad \Gamma \vdash t_2 : W}{\Gamma \vdash t_1\, t_2 : U}$ の場合

  帰納法の仮定より，$\mathrm{RED}_{W \to U}(t_1[x_1, \ldots, x_n := u_1, \ldots, u_n])$ と $\mathrm{RED}_W(t_2[x_1, \ldots, x_n := u_1, \ldots, u_n])$ が成り立つ．このうち前者は，定義を展開すると $\forall v. \mathrm{RED}_W(v) \Rightarrow \mathrm{RED}_U(t_1[x_1, \ldots, x_n := u_1, \ldots, u_n]\, v)$ となるので，$\mathrm{RED}_U((t_1\, t_2)[x_1, \ldots, x_n := u_1, \ldots, u_n])$ が得られる．

- $\dfrac{x_0 : V_0, \Gamma \vdash t' : U'}{\Gamma \vdash (\lambda x_0 : V_0. t') : V_0 \to U'}$ の場合

  任意の $u_0 \in \mathrm{RED}_{V_0}(u_0)$ について $\forall i \in \{0, \ldots, n\}. \mathrm{RED}_{V_i}(u_i)$ が成り立つ．よって，帰納法の仮定より $\mathrm{RED}_{U'}(t'[x_0, x_1, \ldots, x_n := u_0, u_1, \ldots, u_n])$，即ち $\mathrm{RED}_{U'}(t'[x_1, \ldots, x_n := u_1, \ldots, u_n][x_0 := u_0])$ が得られる．

  以上の結果と補題 6.5 より，$\mathrm{RED}_{V_0 \to U'}((\lambda x_0 : V_0. t')[x_1, \ldots, x_n := u_1, \ldots, u_n])$ が示せる． □

以上の事実から，強正規化定理が証明できる．

定理 6.7 (単純型付き λ 計算の強正規化定理) $\Gamma \vdash t : U$ ならば，$\mathrm{SN}(t)$ である．

Proof. $\Gamma = x_1 : V_1, \ldots, x_n : V_n$ とする．CR_3 より変数は任意の型について reducible なので，定理 6.6 より $\mathrm{RED}_U(t[x_1, \ldots, x_n := x_1, \ldots, x_n])$ が成り立つ．この代入は恒等関数なので，$\mathrm{RED}_U(t)$ が成り立つ．CR_1 より，t は強正規化可能である． □

この証明における代入する項を適当な変数 $y_1, \ldots, y_n$ としても，ほぼ同様にして証明できる．その場合は CR_1 より $\mathrm{SN}(t[x_1, \ldots, x_n := y_1, \ldots y_n])$ であり，補題 2.24 より $\mathrm{SN}(t)$ である．

6.2 形式化

本節では，de Bruijn 表現を使う場合に強正規化定理の証明がどのようになるかを見る．まず，reducibility, neutrality の定義，$\mathrm{CR}_{1,2,3}$，補題 6.5 は以下のようになる．これらについては，補題 6.5 で代入の位置が 0 になる以外はほとんど手を加える必要は無い．証明も前節で示したものとあまり変わらないので，省略する．

定義 6.8 (Reducibility) 型 U について，項の集合 RED_U を以下のように定義する．

$$\begin{aligned}\mathrm{RED}_X(t) &\overset{\mathrm{def}}{\iff} \mathrm{SN}(t)\\ \mathrm{RED}_{U\to V}(t) &\overset{\mathrm{def}}{\iff} \forall u.\, \mathrm{RED}_U(u) \Rightarrow \mathrm{RED}_V(t\,u)\end{aligned}$$

定義 6.9 (Neutrality) t が変数か関数適用のどちらかであるとき，項 t は *neutral* であるという．

$$\mathrm{neutral}(t) = \begin{cases}\mathrm{false} & \text{if } t = \lambda : U.\, t'\\ \mathrm{true} & \text{otherwise}\end{cases}$$

補題 6.10 (CR) 以下が成り立つ．

$$\mathrm{RED}_U(t) \Rightarrow \mathrm{SN}(t) \qquad (\mathrm{CR}_1)$$
$$t \to_\beta t' \wedge \mathrm{RED}_U(t) \Rightarrow \mathrm{RED}_U(t') \qquad (\mathrm{CR}_2)$$
$$\mathrm{neutral}(t) \wedge (\forall t'.\, t \to_\beta t' \Rightarrow \mathrm{RED}_U(t')) \Rightarrow \mathrm{RED}_U(t) \qquad (\mathrm{CR}_3)$$

補題 6.11 $\forall v.\, \mathrm{RED}_U(v) \Rightarrow \mathrm{RED}_V(t[0 := v])$ ならば，$\mathrm{RED}_{U\to V}(\lambda : U.\, t)$ が成り立つ．

定理 6.6 は，以下のように少し変形して証明した．

定理 6.12 (Reducibility) 項と型の組の列 $\overline{p}$ について

- $\forall (u, V) \in \overline{p}.\, \mathrm{RED}_V(u)$
- $\lfloor \mathrm{unzip}_2(\overline{p}) \rfloor +\!\!+ \Gamma \vdash t : U$

ならば，$\mathrm{RED}_U(t[0 := \mathrm{unzip}_1(\overline{p})])$ が成り立つ．

Proof. 項 t に関する帰納法で証明する．ただし，その場合分けは定理 6.6 の証明と同様にして $\lfloor \mathrm{unzip}_2(\overline{p}) \rfloor +\!\!+ \Gamma \vdash t : U$ の証明に関する場合分けのように書く．

- $\dfrac{(\lfloor \mathrm{unzip}_2(\overline{p}) \rfloor +\!\!+ \Gamma)_x = \lfloor U \rfloor}{\lfloor \mathrm{unzip}_2(\overline{p}) \rfloor +\!\!+ \Gamma \vdash x : U}$ の場合

  $x < |\overline{p}|$ の場合，仮定の $(\lfloor \mathrm{unzip}_2(\overline{p}) \rfloor +\!\!+ \Gamma)_x = \lfloor U \rfloor$ は

  $$\begin{aligned}(\lfloor \mathrm{unzip}_2(\overline{p}) \rfloor +\!\!+ \Gamma)_x = \lfloor U \rfloor &\Leftrightarrow \lfloor \mathrm{unzip}_2(\overline{p}) \rfloor_x = \lfloor U \rfloor\\ &\Leftrightarrow \lfloor \mathrm{snd}(\overline{p}_x) \rfloor = \lfloor U \rfloor\\ &\Leftrightarrow \mathrm{snd}(\overline{p}_x) = U \qquad (\lfloor \cdot \rfloor \text{ は単射})\end{aligned}$$

と変形できる．これを利用すると

$$\begin{aligned}\mathrm{RED}_U(x[0 := \mathrm{unzip}_1(\overline{p})]) &\Leftrightarrow \mathrm{RED}_{\mathrm{snd}(\overline{p}_x)}(x[0 := \mathrm{unzip}_1(\overline{p})]) && (U \text{ を置換})\\ &\Leftrightarrow \mathrm{RED}_{\mathrm{snd}(\overline{p}_x)}(\mathrm{fst}(\overline{p}_x)\uparrow^0) && (\text{代入の定義})\\ &\Leftrightarrow \mathrm{RED}_{\mathrm{snd}(\overline{p}_x)}(\mathrm{fst}(\overline{p}_x)) && (\uparrow^0 \text{ は恒等関数})\end{aligned}$$

となるので，仮定 $\forall(u,V) \in \overline{p}.\,\mathrm{RED}_V(u)$ より明らか．

$|\overline{p}| \leq x$ の場合，$x[0 := \mathrm{unzip}_1(\overline{p})] = x - |\overline{p}|$ となるので，CR_3 より $\mathrm{RED}_U(x[0 := \mathrm{unzip}_1(\overline{p})])$ が成り立つ．

- $\dfrac{\lfloor \mathrm{unzip}_2(\overline{p})\rfloor +\!\!+ \Gamma \vdash t_1 : V \to U \qquad \lfloor \mathrm{unzip}_2(\overline{p})\rfloor +\!\!+ \Gamma \vdash t_2 : V}{\lfloor \mathrm{unzip}_2(\overline{p})\rfloor +\!\!+ \Gamma \vdash t_1\, t_2 : U}$ の場合

  帰納法の仮定より，$\mathrm{RED}_{V\to W}(t_1[0 := \mathrm{unzip}_1(\overline{p})])$ と $\mathrm{RED}_V(t_2[0 := \mathrm{unzip}_1(\overline{p})])$ が成り立つ．前者は定義を展開すると $\forall v.\,\mathrm{RED}_V(v) \Rightarrow \mathrm{RED}_U(t_1[0 := \mathrm{unzip}_1(\overline{p})]\, v)$ となるので，$\mathrm{RED}_U((t_1\, t_2)[0 := \mathrm{unzip}_1(\overline{p})])$ が成り立つ．

- $\dfrac{\lfloor V_0\rfloor :: \lfloor \mathrm{unzip}_2(\overline{p})\rfloor +\!\!+ \Gamma \vdash t' : U'}{\lfloor \mathrm{unzip}_2(\overline{p})\rfloor +\!\!+ \Gamma \vdash (\lambda : V_0.\, t') : V_0 \to U'}$ の場合

  任意の $u_0 \in \mathrm{RED}_{V_0}$ について，$\overline{p'}$ を $\overline{p'} = (u_0, V_0) :: \overline{p}$ とすると

  - $\forall(u,V) \in \overline{p'}.\,\mathrm{RED}_V(u)$
  - $\lfloor \mathrm{unzip}_2(\overline{p'})\rfloor +\!\!+ \Gamma \vdash t' : U'$

  が成り立つ．よって，帰納法の仮定より $\mathrm{RED}_{U'}(t'[0 := u_0 :: \mathrm{unzip}_1(\overline{p})])$，即ち $\mathrm{RED}_{U'}(t'[1 := \mathrm{unzip}_1(\overline{p})][0 := u_0])$ が得られる[*1]．

  以上の結果と補題 6.11 より，$\mathrm{RED}_{V_0\to U'}(\lambda : V_0.\, t'[1 := \mathrm{unzip}_1(\overline{p})])$ が示せる．代入の定義より，この命題は $\mathrm{RED}_{V_0\to U'}((\lambda : V_0.\, t')[0 := \mathrm{unzip}_1(\overline{p})])$ と同値である． □

定理 6.13 (単純型付き λ 計算の強正規化定理) $\Gamma \vdash t : U$ ならば，$\mathrm{SN}(t)$ である．

Proof. $\overline{p}$ を空列とすると，仮定 $\Gamma \vdash t : U$ と定理 6.12 より $\mathrm{RED}_U(t[0 := \mathrm{unzip}_1([])])$ が成り立つ．CR_1 より $t[0 := \mathrm{unzip}_1([])]$ は強正規化可能である．代入は簡約を保存するので，補題 2.24 より t は強正規化可能である． □

定理 6.12 の仮定にある $+\!\!+\,\Gamma$ は元になっている定理 6.6 には無かった部分であり，一見不要にも見える．しかし，この部分が無いと Γ 中の $\bot$ を $(x, 0)$，$\lfloor U\rfloor$ を (x, U)[*2]にマップして $\overline{p}$ を作る必要がある．その場合には $\Gamma \vdash t : U$ から $\lfloor \mathrm{unzip}_2(\overline{p})\rfloor \vdash t : U$ を示すのにも補題 (5.9) を使う必要がある．一方，$\lfloor \mathrm{unzip}_2(\overline{p})\rfloor +\!\!+ \Gamma$ の形にした場合には上に示したように簡潔な証明が得られる．

[*1] 代入の合成には補題 (4.8) と同等の性質を用いる．

[*2] x は適当な変数．

6.3 Reducibility の再定義

ここまでに見た強正規化定理の証明では，reducible な項であっても型が付くとは限らなかった．例えば，RED_X の定義は SN であるため，$\lambda x : U.\, x\, x$ のような型の付かない項も RED_X に含まれている．

しかし，Girard の証明 [14, Chapter 6] を含むいくつかの強正規化定理の証明では，RED_U は「型 U を持つ項の集合」となっている．本節では，そのように定義された reducibility を**型付き reducibility** (*typed reducibility*) と呼び，型付き reducibility を用いた単純型付き λ 計算の強正規化定理の証明について述べる．

6.3.1 証明に失敗する例

まず，単純な方法で reducibility を型が付く項の集合に制限し，今までと同じような形で CR を示そうとすると証明に失敗することを示す．ここでは，reducibility を以下の形で制限する．この定義では，RED^{Γ}_U は $\{t \mid \Gamma \vdash t : U\}$ の部分集合となる．

定義 6.14 (Reducibility) 型環境 Γ と型 U について，項の集合 RED^{Γ}_U を以下のように定義する．

$$
\begin{aligned}
\mathrm{RED}'^{\Gamma}_X(t) &\overset{\text{def}}{\iff} \mathrm{SN}_{\to_\beta}(t) \\
\mathrm{RED}'^{\Gamma}_{U\to V}(t) &\overset{\text{def}}{\iff} \forall u.\, \mathrm{RED}^{\Gamma}_U(u) \Rightarrow \mathrm{RED}^{\Gamma}_V(t\, u) \\
\mathrm{RED}^{\Gamma}_U(t) &\overset{\text{def}}{\iff} \Gamma \vdash t : U \wedge \mathrm{RED}'^{\Gamma}_U(t)
\end{aligned}
$$

この型付き reducibility に対して，CR は以下のような形で書けるはずである．

命題 6.15 (CR)

$$
\begin{aligned}
\mathrm{RED}^{\Gamma}_U(t) &\Rightarrow \mathrm{SN}(t) && (\mathrm{CR}_1) \\
t \to_\beta t' \wedge \mathrm{RED}^{\Gamma}_U(t) &\Rightarrow \mathrm{RED}^{\Gamma}_U(t') && (\mathrm{CR}_2) \\
\Gamma \vdash t : U \wedge \mathrm{neutral}(t) \wedge (\forall t'.\, t \to_\beta t' \Rightarrow \mathrm{RED}^{\Gamma}_U(t')) &\Rightarrow \mathrm{RED}^{\Gamma}_U(t) && (\mathrm{CR}_3)
\end{aligned}
$$

しかし，補題 6.4 の CR_1 の証明のうち $U = V \to W$ の場合を見ると，RED_V に属する適当な変数 x の存在を利用して証明を行っている．しかし，定義 6.14 の reducibility については RED^{Γ}_V に属する変数が存在しない場合があるため，この CR_1 は今まで通りの方法で証明することはできない．

一方，Girard の証明で用いられている型付き λ 計算の定義 [14, Section 3.1.2] では，任意の型 U に対してその型を持つ可算無限個の変数 $x^U_0, \ldots, x^U_n, \ldots$ が存在するため，このような問題は起きない．

6.3.2 型付き Reducibility による証明 - 1

第 6.1 節の証明について，以下の事実が分かる．

- 最終的に得られた強正規化定理は定理 6.6 と CR_1 から導かれている．
- 定理 6.6 は型付けの証明に関する帰納法で示されており，その証明の中では (補題 6.5 を介して) $\mathrm{CR}_{1,3}$ の結果が使われている．
- $\mathrm{CR}_{1,3}$ は，型に関する帰納法で示されている．

これらのことから，型付けの証明木に含まれる全ての型付け関係について，その一番右側の型の中に $U \to V$ の形で出現する全ての型 U を集めてくることによって，前節の CR_1 の証明で不足していた型を網羅できると考えられる．証明木のサイズが有限であることからこの型の集合は明らかに有限集合であり，具体的には以下の関数 $\mathcal{F}$ によって証明木から目的としている型の集合を構成できる．

$$\mathcal{F}\left(\frac{\Gamma(x) = U}{\Gamma \vdash x : U}\right) = \mathcal{G}(U)$$

$$\mathcal{F}\left(\frac{P_1 \qquad P_2}{\Gamma \vdash t\,u : V}\right) = \mathcal{G}(V) \cup \mathcal{F}(P_1) \cup \mathcal{F}(P_2)$$

$$\mathcal{F}\left(\frac{P_1}{\Gamma \vdash \lambda x : U.t : U \to V}\right) = \mathcal{G}(U \to V) \cup \mathcal{F}(P_1)$$

$$\mathcal{G}(X) = \emptyset$$

$$\mathcal{G}(U \to V) = \{U\} \cup \mathcal{G}(U) \cup \mathcal{G}(V)$$

この $\mathcal{F}$ と $\mathcal{G}$ を用いて，強正規化定理の証明に必要な各種補題は以下の形で証明できる．

補題 6.16 (CR)

$$\mathcal{G}(U) \subseteq \mathrm{codom}(\Gamma) \wedge \mathrm{RED}^{\Gamma}_{U}(t) \Rightarrow \mathrm{SN}(t) \qquad (\mathrm{CR}_1)$$

$$t \to_\beta t' \wedge \mathrm{RED}^{\Gamma}_{U}(t) \Rightarrow \mathrm{RED}^{\Gamma}_{U}(t') \qquad (\mathrm{CR}_2)$$

$$\mathcal{G}(U) \subseteq \mathrm{codom}(\Gamma) \wedge \Gamma \vdash t : U$$
$$\wedge\, \mathrm{neutral}(t) \wedge (\forall t'.\, t \to_\beta t' \Rightarrow \mathrm{RED}^{\Gamma}_{U}(t')) \Rightarrow \mathrm{RED}^{\Gamma}_{U}(t) \qquad (\mathrm{CR}_3)$$

補題 6.17 $\Gamma \vdash \lambda x : U.t : U \to V$ かつ $\mathcal{G}(U \to V) \subseteq \mathrm{codom}(\Gamma)$ かつ $\forall u.\, \mathrm{RED}^{\Gamma}_{U}(u) \Rightarrow \mathrm{RED}^{\Gamma}_{V}(t[x := u])$ ならば，$\mathrm{RED}^{\Gamma}_{U \to V}(\lambda x : U.t)$ が成り立つ．

定理 6.18 (Reducibility) 変数 $x_1, \ldots, x_n$，型 $V_1, \ldots, V_n$，項 $u_1, \ldots, u_n$，型環境 Γ について，$x_1 : V_1, \ldots, x_n : V_n, \Gamma \vdash t : U$ が成り立つとする．また，この型付けの証明を P とする．

もし，$\forall i \leq n.\, \mathrm{RED}^{\Gamma}_{V_i}(u_i)$ かつ $\mathcal{F}(P) \subseteq \mathrm{codom}(\Gamma)$ ならば，$\mathrm{RED}^{\Gamma}_{U}(t[x_1, \ldots, x_n := u_1, \ldots, u_n])$ が成り立つ．

6.3.3 型付き Reducibility による証明 - 2

前節では，CR_1 の証明で必要になる型を事前に型環境に追加しておくことによって，型付き reducibility による強正規化定理の証明が可能であることを説明した．一方で，途中で必要な型を適宜追加していくような形でも証明できる．そのためには，reducibility を以下のように定義する．

定義 6.19 (Reducibility) 型 U について，型環境と項の組の集合 RED_U を以下のように定義する．

$$\begin{aligned}
\mathrm{RED}'_X(\Gamma, t) &\overset{\mathrm{def}}{\iff} \mathrm{SN}_{\to_\beta}(t) \\
\mathrm{RED}'_{U\to V}(\Gamma, t) &\overset{\mathrm{def}}{\iff} \forall u, \Delta.\, \Gamma \leq \Delta \wedge \mathrm{RED}_U(\Delta, u) \Rightarrow \mathrm{RED}_V(\Delta, t\, u) \\
\mathrm{RED}_U(\Gamma, t) &\overset{\mathrm{def}}{\iff} \Gamma \vdash t : U \wedge \mathrm{RED}'_U(\Gamma, t)
\end{aligned}$$

この定義では，CR_1 の証明の問題になっていた箇所で型環境にフレッシュな変数を追加できるようになっている．強正規化定理の証明に必要な補題は，以下に示す形で証明できた．

補題 6.20 (CR) 以下が成り立つ．

$$\mathrm{RED}_U(\Gamma, t) \Rightarrow \mathrm{SN}(t) \qquad (\mathrm{CR}_1)$$

$$t \to_\beta t' \wedge \mathrm{RED}_U(\Gamma, t) \Rightarrow \mathrm{RED}_U(\Gamma, t') \qquad (\mathrm{CR}_2)$$

$$\Gamma \vdash t : U \wedge \mathrm{neutral}(t) \wedge (\forall t'.\, t \to_\beta t' \Rightarrow \mathrm{RED}_U(\Gamma, t')) \Rightarrow \mathrm{RED}_U(\Gamma, t) \qquad (\mathrm{CR}_3)$$

補題 6.21 $\Gamma \vdash \lambda x : U.\, t : U \to V$ かつ $\forall v, \Delta.\, \Gamma \leq \Delta \wedge \mathrm{RED}_U(\Delta, v) \Rightarrow \mathrm{RED}_V(\Delta, t[x := v])$ ならば，$\mathrm{RED}_{U\to V}(\Gamma, \lambda x : U.\, t)$ が成り立つ．

定理 6.22 (Reducibility) 変数 $x_1, \ldots, x_n$，型 $V_1, \ldots, V_n$，項 $u_1, \ldots, u_n$，型環境 Γ について，

- $\forall i \leq n.\, \mathrm{RED}_{V_i}(\Gamma, u_i)$
- $x_1 : V_1, \ldots, x_n : V_n, \Gamma \vdash t : U$

ならば，$\mathrm{RED}_U(\Gamma, t[x_1, \ldots, x_n := u_1, \ldots, u_n])$ が成り立つ．

第 7 章

System F の強正規化定理

本章では，System F の強正規化定理の証明と，その形式化について述べる．

7.1 証明

本節では，Girard の方法 [13, 14] に沿って System F の強正規化定理を証明する．System F では単純型付き λ 計算と同じような方法で直接的に reducibility を定義することができないので，「reducibility の候補」を定義して，それを用いて reducibility を定義する．

定義 7.1 (Neutrality) t が変数，関数適用，型適用のいずれかであるとき，項 t は *neutral* であるという．

$$\mathrm{neutral}(t) = \begin{cases} \text{false} & \text{if } t = \lambda x : U.\, t' \\ \text{false} & \text{if } t = \Lambda X.\, t' \\ \text{true} & \text{otherwise} \end{cases}$$

上述の「reducibility の候補」は以下のように定義できる．

定義 7.2 (Reducibility Candidate) 項の集合 $\mathcal{R}$ が以下の $\mathrm{CR}_{1,2,3}$ を満たすことを $\mathrm{RC}(\mathcal{R})$ と書き，そのような $\mathcal{R}$ を reducibility candidate と呼ぶ．

$$\forall t.\, \mathcal{R}(t) \Rightarrow \mathrm{SN}(t) \qquad (\mathrm{CR}_1)$$

$$\forall t, t'.\, t \to_\beta t' \wedge \mathcal{R}(t) \Rightarrow \mathcal{R}(t') \qquad (\mathrm{CR}_2)$$

$$\forall t.\, \mathrm{neutral}(t) \wedge (\forall t'.\, t \to_\beta t' \Rightarrow \mathcal{R}(t')) \Rightarrow \mathcal{R}(t) \qquad (\mathrm{CR}_3)$$

補題 7.3 SN は reducibility candidate である．

Proof. SN が $\mathrm{CR}_{1,2,3}$ を満たすことを証明する．

(CR_1) $\forall t.\, \mathrm{SN}(t) \Rightarrow \mathrm{SN}(t)$ となるので明らか．

(CR_2) $\forall t, t'.\, t \to_\beta t' \wedge \mathrm{SN}(t) \Rightarrow \mathrm{SN}(t')$ となるので，補題 2.22 より明らか．

(CR_3) $\forall t.\, \mathrm{neutral}(t) \wedge (\forall t'.\, t \to_\beta t' \Rightarrow \mathrm{SN}(t')) \Rightarrow \mathrm{SN}(t)$ となるので，補題 2.22 より明らか． □

補題 7.4 $\mathcal{R}, \mathcal{S}$ が reducibility candidate ならば，以下 $\mathcal{R} \to \mathcal{S}$ は reducibility candidate である．

$$\mathcal{R} \to \mathcal{S} = \{t \mid \forall u.\, \mathcal{R}(u) \Rightarrow \mathcal{S}(t\,u)\}$$

Proof. $\mathcal{R} \to \mathcal{S}$ が $\mathrm{CR}_{1,2,3}$ を満たすことを証明する．

(CR_1) $t \in \mathcal{R} \to \mathcal{S} \Rightarrow \mathrm{SN}(t)$ を示す．

$\mathcal{R}$ についての CR_3 より，フレッシュな変数 x について $\mathcal{R}(x)$ が成り立つ．仮定 $t \in \mathcal{R} \to \mathcal{S}$ より，$\mathcal{S}(t\,x)$ が成り立つ．$\mathcal{S}$ についての CR_1 より，$t\,x$ は強正規化可能である．補題 2.24 より，t は強正規化可能である．

(CR_2) $t \to_\beta t' \wedge t \in \mathcal{R} \to \mathcal{S} \Rightarrow (\forall u.\, \mathcal{R}(u) \Rightarrow \mathcal{S}(t'\,u))$ を示す．

仮定 $\mathcal{R}(u)$ と $t \in \mathcal{R} \to \mathcal{S}$ より，$\mathcal{S}(t\,u)$ が成り立つ．$t \to_\beta t'$ と $\mathcal{S}$ についての CR_2 より，$\mathcal{S}(t'\,u)$ が成り立つ．

(CR_3) 証明すべき命題は以下のようになる．

$$\begin{aligned}
&\mathrm{neutral}(t) && \text{(仮定 1)}\\
&\wedge (\forall t', u'.\, t \to_\beta t' \wedge \mathcal{R}(u') \Rightarrow \mathcal{S}(t'\,u')) && \text{(仮定 2)}\\
\Rightarrow\ &\forall u.\, \mathcal{R}(u) && \text{(仮定 3)}\\
\Rightarrow\ &\mathcal{S}(t\,u)
\end{aligned}$$

仮定 3 と $\mathcal{R}$ についての CR_1 より，u は強正規化可能である．以下の形に命題を変形し，強正規化性に関する帰納法で証明する．

$$\begin{aligned}
&\forall u.\, \mathrm{SN}(u)\\
\Rightarrow\ &\mathcal{R}(u) && \text{(仮定 3′)}\\
\Rightarrow\ &\mathcal{S}(t\,u)
\end{aligned}$$

$\forall v.\, t\,u \to_\beta v \Rightarrow \mathcal{S}(v)$ を $t\,u \to_\beta v$ の証明についての場合分けで示す．

- $(\lambda x.\, t')\,u \to_\beta t'[x := u]$ の場合 $(t = \lambda x.\, t', v = t'[x := u])$

  仮定 1 の $\mathrm{neutral}(\lambda x.\, t')$ より，矛盾が導ける．

- $\dfrac{t \to_\beta t'}{t\,u \to_\beta t'\,u}$ の場合 $(v = t'\,u)$

  仮定 2, 3′ より，$\mathcal{S}(t'\,u)$ が成り立つ．

- $\dfrac{u \to_\beta u'}{t\,u \to_\beta t\,u'}$ の場合 $(v = t\,u')$

  仮定 3′ と $\mathcal{R}$ についての CR_2 より，$\mathcal{R}(u')$ が成り立つ．$u \to_\beta u'$ についての帰納法の仮定より，$\mathcal{S}(t\,u')$ が成り立つ．

これで，$\forall v.\, t\,u \to_\beta v \Rightarrow \mathcal{S}(v)$ が示せた．$\mathcal{S}$ についての CR_3 より，$\mathcal{S}(t\,u)$ が成り立つ．

□

定義 7.5 (Reducibility with Parameters) 型 U と型変数の列 $\overline{X}$ と項の集合の列 $\overline{\mathcal{R}}$ について，項の集合 $\mathrm{RED}_U[\overline{X} := \overline{\mathcal{R}}]$ を以下のように定義する．ただし，$\overline{X}$ は U 中の全ての自由型変数を含んでおり，$\overline{X}$ と $\overline{\mathcal{R}}$ の長さは等しいものとする．

$$\mathrm{RED}_{\overline{X}_i}[\overline{X} := \overline{\mathcal{R}}](t) \overset{\text{def}}{\iff} \overline{\mathcal{R}}_i(t) \qquad (\forall j < i.\, \overline{X}_j \neq \overline{X}_i)$$

$$\mathrm{RED}_{U \to V}[\overline{X} := \overline{\mathcal{R}}](t) \overset{\text{def}}{\iff} t \in \mathrm{RED}_U[\overline{X} := \overline{\mathcal{R}}] \to \mathrm{RED}_V[\overline{X} := \overline{\mathcal{R}}]$$

$$\mathrm{RED}_{\Pi Y.\, U}[\overline{X} := \overline{\mathcal{R}}](t) \overset{\text{def}}{\iff} \forall V, \mathcal{S}.\, \mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_U[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}](t\, V)$$

補題 7.6 $\overline{\mathcal{R}}$ が reducibility candidate の列であれば，$\mathrm{RED}_U[\overline{X} := \overline{\mathcal{R}}]$ は reducibility candidate である．

Proof. 型 U に関する帰納法で証明する．

- $U = \overline{X}_i$ $(\forall j < i.\, \overline{X}_j \neq \overline{X}_i)$ の場合

  仮定より $\overline{\mathcal{R}}_i$ が reducibility candidate なので明らか．

- $U = V \to W$ の場合

  帰納法の仮定より，$\mathrm{RED}_U[\overline{X} := \overline{\mathcal{R}}]$ と $\mathrm{RED}_V[\overline{X} := \overline{\mathcal{R}}]$ は reducibility candidate である．よって，補題 7.4 より，$\mathrm{RED}_U[\overline{X} := \overline{\mathcal{R}}] \to \mathrm{RED}_V[\overline{X} := \overline{\mathcal{R}}]$ も reducibility candidate である．

- $U = \Pi Y.\, U'$ の場合

  (CR_1) $(\forall V, \mathcal{S}.\, \mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_{U'}[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}](t\, V)) \Rightarrow \mathrm{SN}(t)$ を示す．
  仮定の $V, \mathcal{S}$ をそれぞれフレッシュな変数 Z, SN とすると，補題 7.3 より，$\mathrm{RED}_{U'}[Y, \overline{X} := \mathrm{SN}, \overline{\mathcal{R}}](t\, Z)$ が得られる．U' に関する帰納法の仮定 (CR_1) より，$t\, Z$ は強正規化可能である．補題 2.24 より，t は強正規化可能である．

  (CR_2) $t \to_\beta t' \wedge \mathrm{RED}_{\Pi Y.\, U'}[\overline{X} := \overline{\mathcal{R}}](t) \Rightarrow (\forall V\, \mathcal{S}.\, \mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_{U'}[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}](t'\, V))$ を示す．
  仮定の $\mathrm{RED}_{\Pi Y.\, U'}[\overline{X} := \overline{\mathcal{R}}](t)$ と $\mathrm{RC}(\mathcal{S})$ より，$\mathrm{RED}_{U'}[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}](t\, V)$ が成り立つ．U' に関する帰納法の仮定 (CR_2) と $t \to_\beta t'$ より，$\mathrm{RED}_{U'}[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}](t'\, V)$ が成り立つ．

  (CR_3) $\mathrm{neutral}(t) \wedge (\forall t'.\, t \to_\beta t' \Rightarrow \mathrm{RED}_{\Pi Y.\, U'}[\overline{X} := \overline{\mathcal{R}}](t')) \Rightarrow (\forall V, \mathcal{S}.\, \mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_{U'}[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}](t\, V))$ を示す．
  まず，上の論理式に出現している仮定を使って $\forall t_1'.\, t\, V \to_\beta t_1' \Rightarrow \mathrm{RED}_{U'}[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}](t_1')$ を示す．t が neutral であることと $t\, V \to_\beta t_1'$ から，何らかの t_2' について $t_1' = t_2'\, V$ と $t \to_\beta t_2'$ が成り立つ．仮定 $\forall t'.\, t \to_\beta t' \Rightarrow \mathrm{RED}_{\Pi Y.\, U'}[\overline{X} := \overline{\mathcal{R}}](t')$ と $\mathrm{RC}(\mathcal{S})$ より，$\mathrm{RED}_{U'}[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}](t_2'\, V)$ が成り立つ．
  $t\, V$ は neutral なので，直前で得られた結果と U' についての帰納法の仮定 (CR_3) より，$\mathrm{RED}_{U'}[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}](t\, V)$ が成り立つ． □

これ以降では，適当な型 U，U の全ての自由型変数の列 $\overline{X}$，$\overline{X}$ と同じ長さの reducibility

candidate の列 $\overline{\mathcal{R}}$ について，$\mathrm{RED}_U[\overline{\mathcal{R}}](t)$ であることを指して単に「t は reducible である」と言う．補題 7.6 と定義 7.2 の CR_1 より，reducible な項は強正規化可能である．証明の残りの部分では，型の付く項が reducible であることを示す．

補題 7.7 以下が成り立つ．

$$\mathrm{RED}_{U[\overline{Y}:=\overline{V}]}[\overline{X} := \overline{\mathcal{R}}](t) \Leftrightarrow \mathrm{RED}_U[\overline{Y}, \overline{X} := \mathrm{RED}_{\overline{V}}[\overline{X} := \overline{\mathcal{R}}], \overline{\mathcal{R}}](t)$$

ただし，$\mathrm{RED}_{\overline{V}}[\overline{X} := \overline{\mathcal{R}}]$ は $[\mathrm{RED}_V[\overline{X} := \overline{\mathcal{R}}] \mid V \leftarrow \overline{V}]$ の略記である．

Proof. U に関する帰納法で証明する．

- $U = \overline{Y}_i$ の場合

$$\begin{aligned}\mathrm{RED}_{\overline{Y}_i[\overline{Y}:=\overline{V}]}[\overline{X} := \overline{\mathcal{R}}](t) &\Leftrightarrow \mathrm{RED}_{\overline{V}_i}[\overline{X} := \overline{\mathcal{R}}](t)\\ &\Leftrightarrow \mathrm{RED}_{\overline{Y}_i}[\overline{Y}, \overline{X} := \mathrm{RED}_{\overline{V}}[\overline{X} := \overline{\mathcal{R}}], \overline{\mathcal{R}}](t)\end{aligned}$$

- $U = \overline{X}_i (\notin \overline{Y})$ の場合

$$\begin{aligned}\mathrm{RED}_{\overline{X}_i[\overline{Y}:=\overline{V}]}[\overline{X} := \overline{\mathcal{R}}](t) &\Leftrightarrow \mathrm{RED}_{\overline{X}_i}[\overline{X} := \overline{\mathcal{R}}](t)\\ &\Leftrightarrow \overline{\mathcal{R}}_i(t)\\ &\Leftrightarrow \mathrm{RED}_{\overline{X}_i}[\overline{Y}, \overline{X} := \mathrm{RED}_{\overline{V}}[\overline{X} := \overline{\mathcal{R}}], \overline{\mathcal{R}}](t)\end{aligned}$$

- $U = U_1 \to U_2$ の場合

$$\begin{aligned}&\mathrm{RED}_{(U_1\to U_2)[\overline{Y}:=\overline{V}]}[\overline{X} := \overline{\mathcal{R}}](t)\\ \Leftrightarrow\ &\mathrm{RED}_{(U_1[\overline{Y}:=\overline{V}])\to(U_2[\overline{Y}:=\overline{V}])}[\overline{X} := \overline{\mathcal{R}}](t)\\ \Leftrightarrow\ &\forall u.\, \mathrm{RED}_{(U_1[\overline{Y}:=\overline{V}])}[\overline{X} := \overline{\mathcal{R}}](u) \Rightarrow \mathrm{RED}_{(U_1[\overline{Y}:=\overline{V}])}[\overline{X} := \overline{\mathcal{R}}](t\,u)\\ \Leftrightarrow\ &\forall u.\, \mathrm{RED}_{U_1}[\overline{Y}, \overline{X} := \mathrm{RED}_{\overline{V}}[\overline{X} := \overline{\mathcal{R}}], \overline{\mathcal{R}}](u)\\ &\quad \Rightarrow \mathrm{RED}_{(U_1[\overline{Y}:=\overline{V}])}[\overline{X} := \overline{\mathcal{R}}](t\,u) && \text{(I.H. of } U_1\text{)}\\ \Leftrightarrow\ &\forall u.\, \mathrm{RED}_{U_1}[\overline{Y}, \overline{X} := \mathrm{RED}_{\overline{V}}[\overline{X} := \overline{\mathcal{R}}], \overline{\mathcal{R}}](u)\\ &\quad \Rightarrow \mathrm{RED}_{U_2}[\overline{Y}, \overline{X} := \mathrm{RED}_{\overline{V}}[\overline{X} := \overline{\mathcal{R}}], \overline{\mathcal{R}}](t\,u) && \text{(I.H. of } U_2\text{)}\\ \Leftrightarrow\ &\mathrm{RED}_{U_1\to U_2}[\overline{Y}, \overline{X} := \mathrm{RED}_{\overline{V}}[\overline{X} := \overline{\mathcal{R}}], \overline{\mathcal{R}}](t)\end{aligned}$$

- $U = \Pi Z.\, U'$ の場合

$$\begin{aligned}&\mathrm{RED}_{(\Pi Z.\, U')[\overline{Y}:=\overline{V}]}[\overline{X} := \overline{\mathcal{R}}](t)\\ \Leftrightarrow\ &\forall W, \mathcal{S}.\, \mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_{U'[\overline{Y}:=\overline{V}]}[Z, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}](t\,W)\\ \Leftrightarrow\ &\forall W, \mathcal{S}.\, \mathrm{RC}(\mathcal{S})\\ &\qquad \Rightarrow \mathrm{RED}_{U'}[\overline{Y}, Z, \overline{X} := \mathrm{RED}_{\overline{V}}[Z, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}], \mathcal{S}, \overline{\mathcal{R}}](t\,W) && \text{(I.H. of } U'\text{)}\\ \Leftrightarrow\ &\forall W, \mathcal{S}.\, \mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_{U'}[\overline{Y}, Z, \overline{X} := \mathrm{RED}_{\overline{V}}[\overline{X} := \overline{\mathcal{R}}], \mathcal{S}, \overline{\mathcal{R}}](t\,W) && (Z \notin \mathrm{FV}(\overline{V}))\\ \Leftrightarrow\ &\forall W, \mathcal{S}.\, \mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_{U'}[Z, \overline{Y}, \overline{X} := \mathcal{S}, \mathrm{RED}_{\overline{V}}[\overline{X} := \overline{\mathcal{R}}], \overline{\mathcal{R}}](t\,W) && (Z \notin \overline{Y})\\ \Leftrightarrow\ &\mathrm{RED}_{\Pi Z.\, U'}[\overline{Y}, \overline{X} := \mathrm{RED}_{\overline{V}}[\overline{X} := \overline{\mathcal{R}}], \overline{\mathcal{R}}](t)\end{aligned}$$

□

補題 7.8 Reducibility candidate の列 $\overline{\mathcal{R}}$ について

$$\forall u.\, \mathrm{RED}_U[\overline{X} := \overline{\mathcal{R}}](u) \Rightarrow \mathrm{RED}_V[\overline{X} := \overline{\mathcal{R}}](t[x := u])$$

ならば，以下が成り立つ．

$$\mathrm{RED}_{U \to V}[\overline{X} := \overline{\mathcal{R}}](\lambda x : U[\overline{X} := \overline{W}].\, t)$$

Proof. 補題 6.5 の証明とほぼ同様なので，省略する． □

補題 7.9 Reducibility candidate の列 $\overline{\mathcal{R}}$ について $\mathrm{RED}_{\Pi Y.\, U}[\overline{X} := \overline{\mathcal{R}}](t)$ ならば，任意の型 V について以下が成り立つ．

$$\mathrm{RED}_{U[Y:=V]}[\overline{X} := \overline{\mathcal{R}}](t\, V[\overline{X} := \overline{W}])$$

Proof. 仮定と定義 7.5 より，

$$\mathrm{RC}(\mathrm{RED}_V[\overline{X} := \overline{\mathcal{R}}]) \Rightarrow \mathrm{RED}_U[Y, \overline{X} := \mathrm{RED}_V[\overline{X} := \overline{\mathcal{R}}], \overline{\mathcal{R}}](t\, V[\overline{X} := \overline{W}])$$

が成り立つ．補題 7.6 より，上の命題の仮定側にある $\mathrm{RC}(\mathrm{RED}_V[\overline{X} := \overline{\mathcal{R}}])$ は真である．結論側の $\mathrm{RED}_U[Y, \overline{X} := \mathrm{RED}_V[\overline{X} := \overline{\mathcal{R}}], \overline{\mathcal{R}}](t\, V[\overline{X} := \overline{W}])$ は補題 7.7 の右辺と一致するが，左辺の形に変形すると $\mathrm{RED}_{U[Y:=V]}[\overline{X} := \overline{\mathcal{R}}](t\, V[\overline{X} := \overline{W}])$ が得られる． □

補題 7.10 任意の reducibility candidate $\mathcal{R}$ について，以下が成り立つ．

$$\mathcal{R}(t[Y := V]) \Rightarrow \mathcal{R}((\Lambda Y.\, t)\, V)$$

Proof. 仮定と $\mathcal{R}$ についての CR_1 より $t[Y := V]$ は強正規化可能であり，補題 2.24 より t は強正規化可能である．t の強正規化性に関する帰納法で証明を行う．

まず，$\forall u.\, (\Lambda Y.\, t)\, V \rightsquigarrow u \Rightarrow \mathcal{R}(u)$ を $(\Lambda Y.\, t)\, V \rightsquigarrow u$ の証明についての場合分けで示す．

- $(\Lambda Y.\, t)\, V \rightsquigarrow t[Y := V]$ の場合 $(u = t[Y := V])$

  仮定 $\mathcal{R}(t[Y := V])$ より明らか．

- $\dfrac{t \rightsquigarrow t'}{(\Lambda Y.\, t)\, V \rightsquigarrow (\Lambda Y.\, t')\, V}$ の場合 $(u = (\Lambda Y.\, t')\, V)$

  仮定 $\mathcal{R}(t[Y := V])$ と $\mathcal{R}$ についての CR_2 より，$\mathcal{R}(t'[Y := V])$ が成り立つ．よって，$t \rightsquigarrow t'$ についての帰納法の仮定より，$\mathcal{R}((\Lambda Y.\, t')\, V)$ が成り立つ．

これで，$\forall u.\, (\Lambda Y.\, t)\, V \rightsquigarrow u \Rightarrow \mathcal{R}(u)$ が示せた．$\mathcal{R}$ についての CR_3 より，$\mathcal{R}((\Lambda Y.\, t)\, V)$ が成り立つ． □

補題 7.11 Reducibility candidate の列 $\overline{\mathcal{R}}$ について

$$\forall V, \mathcal{S}.\, \mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_U[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}](t[Y := V])$$

ならば，以下が成り立つ．

$$\mathrm{RED}_{\Pi Y.\, U}[\overline{X} := \overline{\mathcal{R}}](\Lambda Y.\, t)$$

Proof. 結論側の命題は，定義 7.5 より $\forall V, \mathcal{S}.\mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_V[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}]((\Lambda Y.\, t)\, V)$ と同値である．よって，任意の型 V と reducibility candidate $\mathcal{S}$ について

$$\mathrm{RED}_U[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}](t[Y := V]) \Rightarrow \mathrm{RED}_U[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}]((\Lambda Y.\, t)\, V)$$

であることを示せば良い．補題 7.6, 7.10 から，これは明らかである． □

定理 7.12 (Reducibility) Reducibility candidate の列 $\overline{\mathcal{R}}$ について

- $\forall i \leq n.\mathrm{RED}_{V_i}[\overline{X} := \overline{\mathcal{R}}](u_i)$
- $x_1 : V_1, \ldots, x_n : V_n \vdash t : U$

ならば，以下が成り立つ．

$$\mathrm{RED}_U[\overline{X} := \overline{\mathcal{R}}](t[\overline{X} := \overline{W}][x_1, \ldots, x_n := u_1, \ldots, u_n])$$

Proof. $x_1 : V_1, \ldots, x_n : V_n$ を Γ と書く．$\Gamma \vdash t : U$ の証明に関する帰納法で示す．

- $\dfrac{\Gamma(y) = U}{\Gamma \vdash y : U}$ の場合
  適当な $i \leq n$ について $y = x_i$ かつ $U = V_i$ が成り立つ．このとき，$y[\overline{X} := \overline{W}][x_1, \ldots, x_n := u_1, \ldots, u_n]$ は u_i となるので，$\mathrm{RED}_{V_i}[\overline{X} := \overline{W}](u_i)$ を示せば良い．これは $\forall i \leq n.\mathrm{RED}_{V_i}[\overline{X} := \overline{W}](u_i)$ より明らか．
- $\dfrac{\Gamma \vdash t_1 : W \to U \qquad \Gamma \vdash t_2 : W}{\Gamma \vdash t_1\, t_2 : U}$ の場合
  帰納法の仮定より，$\mathrm{RED}_{W \to U}[\overline{X} := \overline{W}](t_1[\overline{X} := \overline{W}][x_1, \ldots, x_n := u_1, \ldots, u_n])$ と $\mathrm{RED}_W[\overline{X} := \overline{W}](t_2[\overline{X} := \overline{W}][x_1, \ldots, x_n := u_1, \ldots, u_n])$ が成り立つ．このうち前者は，定義を展開すると $\forall v.\mathrm{RED}_W[\overline{X} := \overline{W}](v) \Rightarrow \mathrm{RED}_U[\overline{X} := \overline{W}](t_1[\overline{X} := \overline{W}][x_1, \ldots, x_n := u_1, \ldots, u_n]\, v)$ となるので，$\mathrm{RED}_U[\overline{X} := \overline{W}]((t_1\, t_2)[\overline{X} := \overline{W}][x_1, \ldots, x_n := u_1, \ldots, u_n])$ が得られる．
- $\dfrac{x_0 : V_0, \Gamma \vdash t' : U'}{\Gamma \vdash \lambda x_0 : V_0.\, t' : V_0 \to U'}$ の場合
  任意の $u_0 \in \mathrm{RED}_{V_0}[\overline{X} := \overline{\mathcal{R}}]$ について $\forall i \in \{0, \ldots, n\}.\mathrm{RED}_{V_i}[\overline{X} := \overline{\mathcal{R}}](u_i)$ が成り立つ．よって，帰納法の仮定より $\mathrm{RED}_{U'}[\overline{X} := \overline{\mathcal{R}}](t'[\overline{X} := \overline{W}][x_0, x_1, \ldots, x_n := u_0, u_1, \ldots, u_n])$，即ち $\mathrm{RED}_{U'}[\overline{X} := \overline{\mathcal{R}}](t'[\overline{X} := \overline{W}][x_1, \ldots, x_n := u_1, \ldots, u_n][x_0 := u_0])$ が得られる．
  以上の結果と補題 7.8 より，$\mathrm{RED}_{V_0 \to U'}[\overline{X} := \overline{\mathcal{R}}]((\lambda x_0 : V_0.\, t')[\overline{X} := \overline{W}][x_1, \ldots, x_n := u_1, \ldots, u_n])$ が示せる．
- $\dfrac{\Gamma \vdash t : \Pi Y.\, U'}{\Gamma \vdash t\, T : U'[Y := T]}$ の場合
  帰納法の仮定 $\mathrm{RED}_{\Pi Y.\, U'}[\overline{X} := \overline{\mathcal{R}}](t[\overline{X} := \overline{W}][x_1, \ldots, x_n := u_1, \ldots, u_n])$ と補題 7.9 より，$\mathrm{RED}_{U'[Y:=T]}[\overline{X} := \overline{\mathcal{R}}](t[\overline{X} := \overline{W}][x_1, \ldots, x_n := u_1, \ldots, u_n]\, T[\overline{X} := \overline{W}])$ が成り立つ．代入の定義より，この命題は今証明しようとしている $\mathrm{RED}_{U'[Y:=T]}[\overline{X} := \overline{\mathcal{R}}]$

$((t\,T)[\overline{X} := \overline{W}][x_1, \ldots, x_n := u_1, \ldots, u_n])$ と同値である．

- $\dfrac{\Gamma \vdash t' : U' \qquad \forall y.\, Y \notin \mathrm{FV}(\Gamma(y))}{\Gamma \vdash \Lambda Y.\, t' : \Pi Y.\, U'}$ の場合

  任意の型 V と reducibility candidate $\mathcal{S}$ について，帰納法の仮定より $\mathrm{RED}_{U'}[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}](t'[Y, \overline{X} := V, \overline{W}][x_1, \ldots, x_n := u_1, \ldots, u_n])$ が成り立つ．Y は $u_1, \ldots, u_n$ 中に出現しないので，この命題は $\mathrm{RED}_{U'}[Y, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}](t'[\overline{X} := \overline{W}][x_1, \ldots, x_n := u_1, \ldots, u_n][Y := V])$ と同値である．

  上の結果と補題 7.11 より，$\mathrm{RED}_{\Pi Y.\, U'}[\overline{X} := \overline{\mathcal{R}}]((\Lambda Y.\, t')[\overline{X} := \overline{W}][x_1, \ldots, x_n := u_1, \ldots, u_n])$ が成り立つ． □

定理 7.13 (System F の強正規化定理) $\Gamma \vdash t : U$ ならば，$\mathrm{SN}(t)$ である．

Proof. $\Gamma = x_1 : V_1, \ldots, x_n : V_n$ として，$u_1, \ldots, u_n, \overline{X}, \overline{\mathcal{R}}, \overline{W}$ を以下のように定める．

- $\forall i \leq n.\, u_i = x_i$
- $\overline{X}$ は $V_1, \ldots, V_n, U$ 中の全ての自由型変数を含む列
- $\overline{\mathcal{R}}$ は $\overline{X}$ と同じ長さの SN の列
- $\overline{W} = \overline{X}$

このとき，補題 7.3 より $\overline{\mathcal{R}}$ は reducibility candidate の列であり，CR_3 より $\forall i \leq n.\, \mathrm{RED}_{V_i}[\overline{X} := \overline{\mathcal{R}}](u_i)$ が成り立つ．よって，定理 7.12 より，$\mathrm{RED}_U[\overline{X} := \overline{\mathcal{R}}](t[\overline{X} := \overline{W}][x_i, \ldots, x_n := u_1, \ldots, u_n])$ が成り立つ．$\overline{W}$ と $u_1, \ldots, u_n$ の選び方から，明らかに $[\overline{X} := \overline{W}]$ と $[x_i, \ldots, x_n := u_1, \ldots, u_n]$ は恒等代入であるため，$\mathrm{RED}_U[\overline{X} := \overline{\mathcal{R}}](t)$ が成り立つ．補題 7.6 と CR_1 より，t は強正規化可能である． □

7.2 形式化

本節では，de Bruijn 表現を使う場合に前節の証明がどのようになるかを見る．

定義 7.14 (Neutrality) t が変数，関数適用，型適用のいずれかであるとき，項 t は *neutral* であるという．

$$\mathrm{neutral}(t) = \begin{cases} \text{false} & \text{if } t = \lambda : U.\, t' \\ \text{false} & \text{if } t = \Lambda\, t' \\ \text{true} & \text{otherwise} \end{cases}$$

定義 7.15 (Reducibility Candidate) 項の集合 $\mathcal{R}$ が以下の $\mathrm{CR}_{1,2,3}$ を満たすことを $\mathrm{RC}(\mathcal{R})$ と書き，そのような $\mathcal{R}$ を reducibility candidate と呼ぶ．

$$\forall t.\, \mathcal{R}(t) \Rightarrow \mathrm{SN}(t) \qquad (\mathrm{CR}_1)$$

$$\forall t, t'.\, t \rightsquigarrow t' \wedge \mathcal{R}(t) \Rightarrow \mathcal{R}(t) \qquad (\mathrm{CR}_2)$$

$$\forall t.\, \mathrm{neutral}(t) \wedge (\forall t'.\, t \rightsquigarrow t' \Rightarrow \mathcal{R}(t')) \Rightarrow \mathcal{R}(t) \qquad (\mathrm{CR}_3)$$

補題 7.16 SN は reducibility candidate である．

補題 7.17 $\mathcal{R}, \mathcal{S}$ が reducibility candidate ならば，以下の $\mathcal{R} \to \mathcal{S}$ は reducibility candidate である．

$$\mathcal{R} \to \mathcal{S} = \{t \mid \forall u.\, \mathcal{R}(u) \Rightarrow \mathcal{S}(t\,u)\}$$

定義 7.18 (Reducibility with Parameters) 型 U と項の集合の列 $\overline{\mathcal{R}}$ について，項の集合 $\mathrm{RED}_U[\overline{\mathcal{R}}]$ を以下のように定義する．

$$\mathrm{RED}_X[\overline{\mathcal{R}}](t) \overset{\mathrm{def}}{\iff} \begin{cases} \overline{\mathcal{R}}_X(t) & \text{if } X < |\overline{\mathcal{R}}| \\ \mathrm{SN}(t) & \text{if } |\overline{\mathcal{R}}| \leq X \end{cases}$$

$$\mathrm{RED}_{U \to V}[\overline{\mathcal{R}}](t) \overset{\mathrm{def}}{\iff} t \in \mathrm{RED}_U[\overline{\mathcal{R}}] \to \mathrm{RED}_V[\overline{\mathcal{R}}]$$

$$\mathrm{RED}_{\Pi Y.\, U}[\overline{\mathcal{R}}](t) \overset{\mathrm{def}}{\iff} \forall V, \mathcal{S}.\, \mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_U[\mathcal{S}, \overline{\mathcal{R}}](t\,V)$$

補題 7.19 もし $\overline{\mathcal{R}}$ が reducibility candidate の列であれば，$\mathrm{RED}_U[\overline{\mathcal{R}}]$ は reducibility candidate である．

補題 7.20 以下が成り立つ．

$$C \leq |\overline{\mathcal{R}}| \Rightarrow \mathrm{RED}_{U \Uparrow_C^{|\overline{\mathcal{S}}|}}[\mathrm{insert}(\overline{\mathcal{S}}, \overline{\mathcal{R}}, C)](t) \Leftrightarrow \mathrm{RED}_U[\overline{\mathcal{R}}](t)$$

補題 7.21 以下が成り立つ．

$$X \leq |\overline{\mathcal{R}}| \Rightarrow \mathrm{RED}_{U[X:=\overline{V}]}[\overline{\mathcal{R}}](t) \Leftrightarrow \mathrm{RED}_U[\mathrm{insert}(\mathrm{RED}_{\overline{V}}[\mathrm{drop}(X, \overline{\mathcal{R}})], \overline{\mathcal{R}}, \mathrm{SN}, X)](t)$$

補題 7.22 Reducibility candidate の列 $\overline{\mathcal{R}}$ について

$$\forall u.\, \mathrm{RED}_U[\overline{\mathcal{R}}](u) \Rightarrow \mathrm{RED}[\overline{\mathcal{R}}](t[0, 0 := u])$$

ならば，以下が成り立つ．

$$\mathrm{RED}_{U \to V}[\overline{\mathcal{R}}](\lambda : U[0 := \overline{W}].\, t)$$

補題 7.23 Reducibility candidate の列 $\overline{\mathcal{R}}$ について $\mathrm{RED}_{\Pi\, U}[\overline{\mathcal{R}}](t)$ ならば，任意の型 V について以下が成り立つ．

$$\mathrm{RED}_{U[0:=V]}[\overline{\mathcal{R}}](t\,V[0 := \overline{W}])$$

補題 7.24 Reducibility candidate の列 $\overline{\mathcal{R}}$ について

$$\forall V, \mathcal{S}.\, \mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_U[\mathcal{S}, \overline{\mathcal{R}}](t[0 := V])$$

ならば，以下が成り立つ．

$$\mathrm{RED}_{\Pi\, U}[\overline{\mathcal{R}}](\Lambda\, t)$$

定理 7.25 (Reducibility) 項の列 $\overline{t}$，型の列 Γ，型の列 $\overline{U}$，reducibility candidate の列 $\overline{\mathcal{R}}$ について，$|\overline{t}| = |\Gamma| = n$, $|\overline{U}| = |\overline{\mathcal{R}}|$ とする．このとき，

$$\lfloor \Gamma \rfloor \vdash u : V \wedge (\forall i < n.\, \mathrm{RED}_{\Gamma_i}[\overline{\mathcal{R}}](\overline{t}_i)) \Rightarrow \mathrm{RED}_V[\overline{\mathcal{R}}](u[0 := \overline{U}][0, 0 := \overline{t}])$$

が成り立つ．

定理 7.26 (System F の強正規化定理) $\Gamma \vdash t : U$ ならば，$\mathrm{SN}(t)$ である．

7.3 Reducibility の再定義 - 1

本節と次節では，第 6.3 節と同様に型付き reducibility を用いた強正規化定理の証明について検討する．

System F の場合には，第 6.3.2 節で述べたような方法で CR_1 の証明に必要な型の有限集合を構成することはできない．しかし，型 $\Pi X.\, X$ を持つ変数 b の存在を仮定すると，任意の型 U に対して $b\,U$ は neutral かつ正規形であり，型 U を持つ．そこで，本節では

$$\#\Gamma = b : \Pi X.\, X, \Gamma$$

という定義を導入し，証明の至る所で型環境から型 $\Pi X.\, X$ を持つ変数 b が取り出せるようにする．この証明方法は，Heq [18] という Coq で heterogeneous equality を扱うためのライブラリの適用例として示されている System F の強正規化定理の証明でも用いられている．

定義 7.27 (Reducibility Candidate) 項の集合 $\mathcal{R}$ が以下の条件を全て満たすことを $\mathrm{RC}^{\Gamma}_{U}(\mathcal{R})$ と書き，そのような $\mathcal{R}$ を Γ, U の reducibility candidate と呼ぶ．

$$\forall t.\, \mathcal{R}(t) \Rightarrow \#\Gamma \vdash t : U \qquad (\mathrm{CR}_{\mathrm{typed}})$$
$$\forall t.\, \mathcal{R}(t) \Rightarrow \mathrm{SN}(t) \qquad (\mathrm{CR}_1)$$
$$\forall t, t'.\, t \to_\beta t' \wedge \mathcal{R}(t) \Rightarrow \mathcal{R}(t') \qquad (\mathrm{CR}_2)$$
$$\forall t.\, \#\Gamma \vdash t : U \wedge \mathrm{neutral}(t) \wedge (\forall t'.\, t \to_\beta t' \Rightarrow \mathcal{R}(t')) \Rightarrow \mathcal{R}(t) \qquad (\mathrm{CR}_3)$$

定義 7.28 (Reducibility with Parameters) 型 V と型変数の列 $\overline{X}$ と型の列 $\overline{U}$ と項の集合の列 $\overline{\mathcal{R}}$ について，項の集合 $\mathrm{RED}^{\Gamma}_{V}[\overline{X} : \overline{U} := \overline{\mathcal{R}}]$ を以下のように定義する．ただし，$\overline{X}$ は V 中の全ての自由型変数を含んでおり，$\overline{X}$ と $\overline{U}$ と $\overline{\mathcal{R}}$ の長さは等しいものとする．

$$\mathrm{RED}'^{\Gamma}_{X_i}[\overline{X} : \overline{U} := \overline{\mathcal{R}}](t) \overset{\mathrm{def}}{\iff} \overline{\mathcal{R}}_i(t) \qquad (\forall j < i.\, \overline{X}_j \neq \overline{X}_i)$$

$$\mathrm{RED}'^{\Gamma}_{V \to W}[\overline{X} : \overline{U} := \overline{\mathcal{R}}](t) \overset{\mathrm{def}}{\iff} \forall u.\, \mathrm{RED}^{\Gamma}_{V}[\overline{X} : \overline{U} := \overline{\mathcal{R}}](u) \Rightarrow \mathrm{RED}^{\Gamma}_{W}[\overline{X} : \overline{U} := \overline{\mathcal{R}}](t\,u)$$

$$\mathrm{RED}'^{\Gamma}_{\Pi Y.\, V}[\overline{X} : \overline{U} := \overline{\mathcal{R}}](t) \overset{\mathrm{def}}{\iff} \forall W, \mathcal{S}.\, \mathrm{RC}^{\Gamma}_{W}(\mathcal{S}) \Rightarrow \mathrm{RED}^{\Gamma}_{V}[Y, \overline{X} : W, \overline{U} := \mathcal{S}, \overline{\mathcal{R}}](t\,W)$$

$$\mathrm{RED}^{\Gamma}_{V}[\overline{X} : \overline{U} := \overline{\mathcal{R}}](t) \overset{\mathrm{def}}{\iff} \#\Gamma \vdash t : V[\overline{X} := \overline{U}] \wedge \mathrm{RED}'^{\Gamma}_{V}[\overline{X} : \overline{U} := \overline{\mathcal{R}}](t)$$

補題 7.29 $\forall i.\, \mathrm{RC}^{\Gamma}_{\overline{U}_i}(\overline{\mathcal{R}}_i)$ ならば，$\mathrm{RC}^{\Gamma}_{V[\overline{X} := \overline{U}]}(\mathrm{RED}^{\Gamma}_{V}[\overline{X} : \overline{U} := \overline{\mathcal{R}}])$ が成り立つ．

定理 7.30 (Reducibility)

- $\forall i.\, \mathrm{RC}^{\Gamma}_{\overline{U}_i}(\overline{\mathcal{R}}_i)$
- $\forall i \leq n.\, \mathrm{RED}^{\Gamma}_{W_i}[\overline{X} : \overline{U} := \overline{\mathcal{R}}](u_i)$
- $x_1 : W_1, \ldots, x_n : W_n \vdash t : V$

ならば，以下が成り立つ．

$$\mathrm{RED}^{\Gamma}_{V}[\overline{X} : \overline{U} := \overline{\mathcal{R}}](t[\overline{X} := \overline{U}][x_1, \ldots, x_n := u_1, \ldots, u_n])$$

7.4 Reducibility の再定義 - 2

本節では，Girard の証明を元にして Gallier の証明 [11, Section 9] を参考に変更を加えて得られた証明を紹介する．この証明では，第 6.3.3 節で示した証明に近い方法を取っている．

定義 7.31 (Reducibility Candidate) 型環境と項の組の集合 $\mathcal{R}$ が以下の条件を全て満たすことを $\mathrm{RC}_U(\mathcal{R})$ と書き，そのような $\mathcal{R}$ を U の reducibility candidate と呼ぶ．

$$\forall \Gamma, t.\, \mathcal{R}(\Gamma, t) \Rightarrow \Gamma \vdash t : U \qquad (\mathrm{CR}_{\mathrm{typed}})$$
$$\forall \Gamma, \Delta, t.\, \Gamma \leq \Delta \wedge \mathcal{R}(\Gamma, t) \Rightarrow \mathcal{R}(\Delta, t) \qquad (\mathrm{CR}_0)$$
$$\forall \Gamma, t.\, \mathcal{R}(\Gamma, t) \Rightarrow \mathrm{SN}(\Gamma, t) \qquad (\mathrm{CR}_1)$$
$$\forall \Gamma, t, t'.\, t \to_\beta t' \wedge \mathcal{R}(\Gamma, t) \Rightarrow \mathcal{R}(\Gamma, t') \qquad (\mathrm{CR}_2)$$
$$\forall \Gamma, t.\, \Gamma \vdash t : U \wedge \mathrm{neutral}(t) \wedge (\forall t'.\, t \to_\beta t' \Rightarrow \mathcal{R}(\Gamma, t')) \Rightarrow \mathcal{R}(\Gamma, t) \qquad (\mathrm{CR}_3)$$

定義 7.32 (Reducibility with Parameters) 型 V と型変数の列 $\overline{X}$ と型の列 $\overline{U}$ と型環境と項の組の集合の列 $\overline{\mathcal{R}}$ について，型環境と項の組の集合 $\mathrm{RED}_V[\overline{X} : \overline{U} := \overline{\mathcal{R}}]$ を以下のように定義する．ただし，$\overline{X}$ は V 中の全ての自由型変数を含んでおり，$\overline{X}$ と $\overline{U}$ と $\overline{\mathcal{R}}$ の長さは等しいものとする．

$$\mathrm{RED}'_{\overline{X}_i}[\overline{X} : \overline{U} := \overline{\mathcal{R}}](\Gamma, t) \overset{\mathrm{def}}{\iff} \overline{\mathcal{R}}_i(\Gamma, t) \qquad (\forall j < i.\, \overline{X}_j \neq \overline{X}_i)$$

$$\begin{aligned}\mathrm{RED}'_{V \to W}[\overline{X} : \overline{U} := \overline{\mathcal{R}}](\Gamma, t) \overset{\mathrm{def}}{\iff} \forall \Delta, u.\, \Gamma \leq \Delta \\ \wedge \mathrm{RED}_V[\overline{X} : \overline{U} := \overline{\mathcal{R}}](\Delta, u) \\ \Rightarrow \mathrm{RED}_W[\overline{X} : \overline{U} := \overline{\mathcal{R}}](\Delta, t\, u)\end{aligned}$$

$$\begin{aligned}\mathrm{RED}'_{\Pi Y.\, V}[\overline{X} : \overline{U} := \overline{\mathcal{R}}](\Gamma, t) \overset{\mathrm{def}}{\iff} \forall W, \mathcal{S}.\, \mathrm{RC}_W(\mathcal{S}) \\ \Rightarrow \mathrm{RED}_V[Y, \overline{X} : W, \overline{U} := \mathcal{S}, \overline{\mathcal{R}}](\Gamma, t\, W)\end{aligned}$$

$$\mathrm{RED}_V[\overline{X} : \overline{U} := \overline{\mathcal{R}}](\Gamma, t) \overset{\mathrm{def}}{\iff} \Gamma \vdash t : V[\overline{X} := \overline{U}] \wedge \mathrm{RED}'_V[\overline{X} : \overline{U} := \overline{\mathcal{R}}](\Gamma, t)$$

補題 7.33 $\forall i.\, \mathrm{RC}_{\overline{U}_i}(\overline{\mathcal{R}}_i)$ ならば，$\mathrm{RC}_{V[\overline{X} := \overline{U}]}(\mathrm{RED}_V[\overline{X} : \overline{U} := \overline{\mathcal{R}}])$ が成り立つ．

定理 7.34 (Reducibility)

- $\forall i.\, \mathrm{RC}_{\overline{U}_i}(\overline{\mathcal{R}}_i)$
- $\forall i \leq n.\, \mathrm{RED}_{W_i}[\overline{X} : \overline{U} := \overline{\mathcal{R}}](\Gamma, u_i)$
- $x_1 : W_1, \ldots, x_n : W_n \vdash t : V$

ならば，以下が成り立つ．

$$\mathrm{RED}_V[\overline{X} : \overline{U} := \overline{\mathcal{R}}](\Gamma, t[\overline{X} := \overline{U}][x_1, \ldots, x_n := u_1, \ldots, u_n])$$

第 8 章

関連研究

本章では，本研究の関連研究を束縛変数の形式化と自動証明に関するもの (第 8.1 節) と強正規化定理の形式化に関するもの (第 8.2 節) に分けて紹介する．

8.1 束縛変数の形式化と自動証明

Schäfer ら [26] は，de Bruijn 表現の項と並列代入[*1]の等価性について健全かつ完全な **de Bruijn 代数**を定義し，de Bruijn 代数の等式が決定可能であることを証明している．その原理を応用して作られた Coq のライブラリが Autosubst [27] である．Autosubst は代入補題などの自動証明だけではなく，項のデータ型の定義から代入の定義や基盤となる補題を自動生成する仕組みなども持っている．同様の自動生成ツールとしては，他にも locally nameless 表現用の LNgen [2] や de Bruijn 表現用の DBGen [23, 24] がある．

Nipkow [21] は，図 3.1 の定義を $d = 1$ の場合に制限したシフトと図 3.2 の代入を用いて，Isabelle/HOL で型無し λ 計算の合流性の証明を行っている．この証明でのシフトと代入に関する補題は，以下のようになっている:

$$
\begin{aligned}
c \le c' \Rightarrow t \uparrow^1_c \uparrow^1_{c'+1} &= t \uparrow^1_{c'} \uparrow^1_c \\
x \le c \Rightarrow t\{x := u\} \uparrow^1_c &= t \uparrow^1_{c+1} \{x := u \uparrow^1_c\} \\
c \le x \Rightarrow t\{x := u\} \uparrow^1_c &= t \uparrow^1_c \{x+1 := u \uparrow^1_c\} \\
t \uparrow^1_x \{x := u\} &= t \\
x \le y \Rightarrow t\{y+1 := v \uparrow^1_x\}\{x := u\{y := v\}\} &= t\{x := u\}\{y := v\}
\end{aligned}
$$

この証明の補題の数は我々の方法と比較して非常に少なく，証明自体もシンプルである．一方で，この論文の第 7 節では図 3.3 の形の代入の定義が実装のための最適化された (比較的証明に向いていない) 定義として紹介されている．Nipkow の方法をそのまま図 3.4 の並列代入に適用しようとすると，代入補題の命題は

$$
x \le y \Rightarrow t\{y + |\overline{u}| := \overline{v} \uparrow^{|\overline{u}|}_x\}\{x := \overline{u}\{y := \overline{v}\}\} = t\{x := \overline{u}\}\{y := \overline{v}\}
$$

[*1] ここでは全ての自由変数を置き換えるような代入を指しており，代入自体は自然数から項への関数で表される．

となる．Nipkow の証明が非常に簡潔な理由としては $d = 1$ のシフトしか扱わずに済んでいるという点が大きいと考えられるが，この命題では $d = |\overline{u}|$ のシフトが出現している．よって，並列代入に拡張した場合にはより一般的な補題が必要になり，証明も複雑になると考えられる．

一方で，図 3.5 の並列代入の定義でシフトを 1 箇所にまとめられているという点に注目して自動化を行ったのが，我々の方法である．図 4.1 の全ての補題の証明について，帰納法を適用した後の関数適用と λ 抽象の場合の証明は $+1$ の付け替えと帰納法の仮定での書き換えだけで証明できる．しかし，図 3.4 の代入の定義では λ 抽象の場合に代入する項をシフトしているため，このような証明法は通用しない．

8.2 強正規化定理の形式化

定理証明器を用いて System F の強正規化定理を形式化した例としては，Altenkirch による LEGO での形式化 [1]，Donnelly らによる ATS/LF での形式化 [10]，Hur による Coq での形式化 [18] が挙げられる．このうち，Altenkirch の形式化と Donnelly らの形式化は第 7.1 節で示した方法に近い証明になっており，Hur の形式化は第 7.3 節で示した方法に近い証明になっている．また，第 7.4 節の方法に基く形式化は現時点では我々の証明を除いて発見できていない．

第 9 章

結論

本研究では，定理証明器 Coq の上で単純型付き λ 計算と System F を定義し，Tait と Girard の方法に基いてそれらの体系の強正規化定理を証明した．本研究を通じて，強正規化定理の証明に使われる reducibility にはいくつかの定義方法があり，それぞれ微妙に異る証明が得られることを明らかにした．また，Girard の証明 [14, Chapter 6, Chapter 14] では型付き λ 計算の定義方法が我々の定義と異なっており，この証明をそのまま我々の定義に適用することはできない．本研究での形式化を通じて，この問題の原因と解決法を明らかにした．

強正規化定理の形式化の基盤として，de Bruijn 表現で定義された項についての基本的な性質を自動的に証明する方法を考案した．強正規化定理の証明には並列代入が必要であり，代入の定義と自動証明の仕組みは証明が上手くいくかどうかという点で相互に影響し合うため，この点については試行錯誤や検討を重ねて決定した．結果的に，`congruence` タクティクや `lia` タクティクなどの Coq が元々持っている機能を最大限生かす形で，代入補題などの証明の多くの部分を自動的に証明する方法を与えられた．

本研究の主な課題としては，より広い範囲の体系について，より多くの方法に基く強正規化定理の証明を形式化することが挙げられる．例えば，System F に拡張を加えた System Fω, Calculus of Constructions (CoC) についての強正規化定理の形式化や，Gandy の方法 [12] に基く証明の形式化にも着手したいと考えている．また，型付き λ 計算のモデルと論理関係 (logical relation) などのより一般的な枠組みでの結果を用いて，再度形式化を行いたい．

自動証明についても，現時点では部分的な自動化に留まっているため，より広い範囲の自動化に取り組みたい．特に，nth の扱いは改善の余地があると考えている．また，それに関連して `omega` と `lia` の性能の比較を行いたい．Coq のリファレンスマニュアル [29, Section 21.5] には以下のような一節があり，このような調査はまだ不十分であることが分かる．

> The estimation of the relative efficiency of `lia` vs `omega` and `romega` is under evaluation.

本研究の事例からも分かる通り，`omega` と `lia` は「自然数の減算が多い」などの論理式の特徴によって大きく性能が変わる．よって，どのような論理式に対してどちらのタクティクが向いているかという指標を与えられる可能性があり，それは多くの Coq ユーザにとって意味のあるタクティク選定の基準となり得る．

謝辞

本研究を行うにあたり，指導教員として多くの指導と助言を頂きました南出靖彦先生に深く感謝します．本研究に関連する多くの助言を頂きました亀山幸義先生 (筑波大学) と浜名誠先生 (群馬大学) に感謝します．Coq と SSReflect に関して多くのことを教えて頂き，また本研究に関連する非常に貴重な文献 [13] を貸して頂きました Reynald Affeldt さん (産業技術総合研究所) に感謝します．本研究に関する貴重な文献 [12] を貸して頂きました竹内泉さん (産業技術総合研究所) に感謝します．本研究は，坂口が平成 24 年度から取り組んでいた問題を元にしており，平成 25 年度の筑波大学先導的研究者体験プログラム (ARE) による支援を受けました．ARE 関係者の皆様に感謝します．

参考文献

[1] Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In J.F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, LNCS 664, pages 13 – 28, 1993.

[2] Brian Aydemir. LNgen: Tool support for locally nameless representations, 2010. URL: http://www.cis.upenn.edu/~sweirich/papers/lngen/.

[3] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 3–15, 2008.

[4] Henk P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. Elsevier, revised edition, 1984.

[5] Nick Benton, Chung-Kil Hur, AndrewJ. Kennedy, and Conor McBride. Strongly typed term representations in Coq. *Journal of Automated Reasoning*, 49(2):141–159, 2012.

[6] Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*. 2007.

[7] Richard S. Bird and Ross Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9:77–91, 1 1999.

[8] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 143–156, 2008.

[9] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972.

[10] Kevin Donnelly and Hongwei Xi. A formalization of strong normalization for simply typed lambda-calculus and System F. In *Proceedings of Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, LFMTP'06, pages 109–125. ENTCS 174(5), 2006.

[11] Jean H. Gallier. On Girard's "candidats de reductibilité". In *Logic and Computer Science*. Academic Press, 1989.

[12] Robin O. Gandy. Proof of strong normalization. In *To H.B. Curry: Essays on Combinatory*

Logic, Lambda Calculus and Formalism. Academic Press, 1980.

[13] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université de Paris 7, 1972.

[14] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[15] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A small scale reflection extension for the Coq system. Research report, INRIA, 2008. URL: http://hal.inria.fr/inria-00258384.

[16] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, 2nd edition, 2008.

[17] Gérard Huet. Residual theory in λ-calculus: a formal development. *Journal of Functional Programming*, 4:371–394, 7 1994.

[18] Chung-Kil Hur. Heq : a Coq library for heterogeneous equality, 2010. URL: http://sf.snu.ac.kr/gil.hur/Heq/.

[19] Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.

[20] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.

[21] Tobias Nipkow. More Church-Rosser proofs. *Journal of Automated Reasoning*, 26(1):51–66, 2001.

[22] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 199–208, 1988.

[23] Emmanuel Polonowski. Automatically generated infrastructure for de bruijn syntaxes. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*. 2013.

[24] Emmanuel Polonowski. DBGen - de Bruijn infrastructure generation for Coq proof assistant, 2013. URL: http://www.lacl.fr/~polonowski/Develop/DBGen/dbgen.html.

[25] Kazuhiko Sakaguchi. A formalization of typed and untyped λ-calculi in SSReflect-Coq and Agda2, 2011-2015. URL: https://github.com/pi8027/lambda-calculus.

[26] Steven Schäfer, Gert Smolka, and Tobias Tebbi. Completeness and decidability of de Bruijn substitution algebra in Coq. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, pages 67–73. ACM, 2015.

[27] Steven Schäfer and Tobias Tebbi. Autosubst: Automation for de Bruijn syntax and substitution in Coq, 2014. URL: https://www.ps.uni-saarland.de/autosubst/.

[28] William W. Tait. Intensional interpretations of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

[29] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2014. URL: https://coq.inria.fr/distrib/V8.4pl5/refman/.

付録 A

ソースコード

A.1 ssrnat_ext.v

```
Require Import
  Ssreflect.ssreflect Ssreflect.ssrfun Ssreflect.ssrbool Ssreflect.eqtype
  Ssreflect.ssrnat Ssreflect.seq Omega Psatz LCAC.lib.seq_ext_base.

Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.

Definition natE :=
  (addSn, addnS, add0n, addn0, sub0n, subn0, subSS,
   min0n, minn0, max0n, maxn0, leq0n).
```

A.1.1 拡張された比較述語

```
CoInductive leq_xor_gtn' m n :
    bool -> bool -> bool -> bool ->
    nat -> nat -> nat -> nat -> nat -> nat -> Set :=
  | LeqNotGtn' of m <= n :
    leq_xor_gtn' m n (m < n) false true (n <= m) n n m m 0 (n - m)
  | GtnNotLeq' of n < m :
    leq_xor_gtn' m n false true false true m m n n (m - n) 0.

Lemma leqP' m n : leq_xor_gtn' m n
  (m < n) (n < m) (m <= n) (n <= m)
  (maxn m n) (maxn n m) (minn m n) (minn n m)
  (m - n) (n - m).
Proof.
  case: (leqP m n) => H; rewrite (maxnC n) (minnC n).
  - rewrite (maxn_idPr H) (minn_idPl H).
    by move: (H); rewrite -subn_eq0 => /eqP ->; constructor.
  - rewrite (ltnW H) ltnNge leq_eqVlt H orbT
            (maxn_idPl (ltnW H)) (minn_idPr (ltnW H)).
    by move: (ltnW H); rewrite -subn_eq0 => /eqP ->; constructor.
Qed.

CoInductive compare_nat' m n :
```

```
    bool -> bool -> bool -> bool -> bool ->
    nat -> nat -> nat -> nat -> nat -> nat -> Set :=
  | CompareNatLt' of m < n :
    compare_nat' m n true false false true false n n m m 0 (n - m)
  | CompareNatGt' of m > n :
    compare_nat' m n false true false false true m m n n (m - n) 0
  | CompareNatEq' of m = n :
    compare_nat' m n false false true true true m m m m 0 0.

Lemma ltngtP' m n : compare_nat' m n
  (m < n) (n < m) (m == n) (m <= n) (n <= m)
  (maxn m n) (maxn n m) (minn m n) (minn n m)
  (m - n) (n - m).
Proof.
  (case: (ltngtP m n) => H;
    last by rewrite -H leqnn maxnn minnn subnn; constructor);
    rewrite (maxnC n) (minnC n) ?(ltnW H) leqNgt H /=.
  - rewrite (maxn_idPr (ltnW H)) (minn_idPl (ltnW H)).
    by move: (ltnW H); rewrite -subn_eq0 => /eqP ->; constructor.
  - rewrite (maxn_idPl (ltnW H)) (minn_idPr (ltnW H)).
    by move: (ltnW H); rewrite -subn_eq0 => /eqP ->; constructor.
Qed.
```

A.1.2 simpl_natarith タクティク

```
Module simpl_natarith.
Lemma lem1_1 ml mr n r : ml = r + n -> ml + mr = r + mr + n.
Proof. by move => ->; rewrite addnAC. Qed.
Lemma lem1_2 ml mr n r : mr = r + n -> ml + mr = ml + r + n.
Proof. by move => ->; rewrite addnA. Qed.
Lemma lem1_3 m' n r : m' = r + n -> m'.+1 = r.+1 + n.
Proof. by move => ->; rewrite addSn. Qed.
Lemma lem2_1 ml mr n r : ml - n = r -> ml - mr - n = r - mr.
Proof. by move => <-; rewrite subnAC. Qed.
Lemma lem2_2 m' n r : m' - n = r -> m'.-1 - n = r.-1.
Proof. by move => <-; rewrite -subnS -add1n subnDA subn1. Qed.
Lemma lem2_3 m n r : m = r + n -> m - n = r.
Proof. by move => ->; rewrite addnK. Qed.
Lemma lem3_1 m m' m'' nl nl' nl'' nr nr' :
  m - nl = m' - nl' -> m' - nl' - nr = m'' - nl'' - nr' ->
  m - (nl + nr) = m'' - (nl'' + nr').
Proof. by rewrite !subnDA => -> ->. Qed.
Lemma lem3_2 m n r : m - (n + 1) = r -> m - n.+1 = r.
Proof. by rewrite addn1. Qed.
Lemma lem3_3 m n r : m - n = r -> m - n = r - 0.
Proof. by rewrite subn0. Qed.
Lemma lem4_1 m n m' n' : m - n = m' - n' -> (m <= n) = (m' <= n').
Proof. by rewrite -!subn_eq0 => ->. Qed.
End simpl_natarith.
Import simpl_natarith.

Ltac simpl_natarith1 m n :=
  match m with
    | n => constr: (esym (add0n n))
```

```
    | ?ml + ?mr => let H := simpl_natarith1 ml n in constr: (lem1_1 mr H)
    | ?ml + ?mr => let H := simpl_natarith1 mr n in constr: (lem1_2 ml H)
    | ?m'.+1 => let H := simpl_natarith1 m' n in constr: (lem1_3 H)
    | ?m'.+1 => match n with 1 => constr: (esym (addn1 m')) end
  end.

Ltac simpl_natarith2 m n :=
  match m with
    | ?ml - ?mr => let H := simpl_natarith2 ml n in constr: (lem2_1 mr H)
    | ?m'.-1 => let H := simpl_natarith2 m' n in constr: (lem2_2 H)
    | _ => let H := simpl_natarith1 m n in constr: (lem2_3 H)
  end.

Ltac simpl_natarith3 m n :=
  lazymatch n with
    | ?nl + ?nr =>
      simpl_natarith3 m nl;
      match goal with |- _ = ?m1 -> _ =>
        let H := fresh "H" in
        move => H; simpl_natarith3 m1 nr; move/(lem3_1 H) => {H}
      end
    | _ =>
      match n with
        | ?n'.+1 =>
          lazymatch n' with
            | 0 => fail
            | _ => simpl_natarith3 m (n' + 1); move/lem3_2
          end
        | _ => let H := simpl_natarith2 m n in move: (lem3_3 H)
        | _ => move: (erefl (m - n))
      end
  end.

Ltac simpl_natarith :=
  let tac x :=
    lazymatch goal with
      | |- ?x = ?x -> _ => move => _; rewrite !natE
      | _ => move => ->; rewrite ?natE
    end in
  repeat
    (match goal with
       H : context [?m - ?n] |- _ => move: H; simpl_natarith3 m n; tac 0 => H
     end ||
     match goal with
       |- context [?m - ?n] => simpl_natarith3 m n; tac 0
     end ||
     match goal with
       H : context [?m <= ?n] |- _ =>
       move: H; simpl_natarith3 m n; move/lem4_1; tac 0 => H
     end ||
     match goal with
       |- context [?m <= ?n] => simpl_natarith3 m n; move/lem4_1; tac 0
     end);
  try done;
  repeat match goal with
    | H : is_true true |- _ => clear H
```

```
  end.
```

A.1.3 elimleq タクティク

```
Tactic Notation "elimleq" :=
  match goal with |- is_true (?n <= ?m) -> _ =>
    is_var m;
    (let H := fresh "H" in move/subnKC => H; rewrite <- H in *; clear H);
    let rec tac :=
      lazymatch reverse goal with
        | H : context [m] |- _ => move: H; tac => H
        | _ => move: {m} (m - n) => m; rewrite ?(addKn, addnK)
      end in tac; simpl_natarith
  end.

Tactic Notation "elimleq" constr(H) := move: H; elimleq.
```

A.1.4 ssromega タクティク

```
Tactic Notation "find_minneq_hyp" constr(m) constr(n) :=
  match goal with
    | H : is_true (m <= n) |- _ => rewrite (minn_idPl H)
    | H : is_true (n <= m) |- _ => rewrite (minn_idPr H)
    | H : is_true (m < n) |- _ => rewrite (minn_idPl (ltnW H))
    | H : is_true (n < m) |- _ => rewrite (minn_idPr (ltnW H))
    | |- _ => case (leqP' m n)
  end; rewrite ?natE.

Tactic Notation "find_maxneq_hyp" constr(m) constr(n) :=
  match goal with
    | H : is_true (m <= n) |- _ => rewrite (maxn_idPr H)
    | H : is_true (n <= m) |- _ => rewrite (maxn_idPl H)
    | H : is_true (m < n) |- _ => rewrite (maxn_idPr (ltnW H))
    | H : is_true (n < m) |- _ => rewrite (maxn_idPl (ltnW H))
    | |- _ => case (leqP' m n)
  end; rewrite ?natE.

Ltac replace_minn_maxn :=
  try (rewrite <- minnE in * || rewrite <- maxnE in * );
  match goal with
    | H : context [minn ?m ?n] |- _ => move: H; find_minneq_hyp n m => H
    | H : context [maxn ?m ?n] |- _ => move: H; find_maxneq_hyp n m => H
    | |- context [minn ?m ?n] => find_minneq_hyp m n
    | |- context [maxn ?m ?n] => find_maxneq_hyp m n
  end;
  try (let x := fresh "x" in move => x).

Ltac arith_hypo_ssrnat2coqnat :=
  match goal with
    | H : is_true false    |- _ => by move: H
    | H : is_true (_ && _) |- _ => let H0 := fresh "H" in case/andP: H => H H0
    | H : is_true (_ || _) |- _ => case/orP in H
```

```
    | H : is_true (_ <  _) |- _ => move/ltP in H
    | H : is_true (_ <= _) |- _ => move/leP in H
    | H : is_true (_ == _) |- _ => move/eqP in H
  end.

Ltac arith_goal_ssrnat2coqnat :=
  match goal with
    | |- is_true (_ && _) => apply/andP; split
    | |- is_true (_ || _) => apply/orP
    | |- is_true (_ <  _) => apply/ltP
    | |- is_true (_ <= _) => apply/leP
    | |- is_true (_ == _) => apply/eqP
  end.

Ltac ssromega :=
  repeat (let x := fresh "x" in move => x);
  do ?replace_minn_maxn;
  try done;
  repeat match goal with H : is_true (?m <= ?n) |- _ => elimleq H end;
  do ?unfold addn, subn, muln, addn_rec, subn_rec, muln_rec in *;
  do ?arith_hypo_ssrnat2coqnat;
  do ?arith_goal_ssrnat2coqnat;
  simpl Equality.sort in *;
  lia.
```

A.1.5 elimif_omega タクティク

```
Ltac elimif' :=
  (match goal with
     | |- context [if ?m < ?n then _ else _] => case (leqP' n m)
     | |- context [if ?m <= ?n then _ else _] => case (leqP' m n)
     | |- context [if ?b then _ else _] => case (ifP b)
   end;
   move => //=; elimif'; let hyp := fresh "H" in move => hyp) ||
  idtac.

Ltac elimif :=
  elimif'; simpl_natarith;
  repeat match goal with H : is_true (?m <= ?n) |- _ => elimleq H end.

Ltac elimif_omega :=
  elimif;
  try (repeat match goal with
    | |- @eq nat _ _ => idtac
    | |- nth _ _ _ = nth _ _ _ => apply nth_equal
    | |- _ => f_equal
  end; ssromega).
```

A.1.6 congruence' タクティク

```
Ltac congruence' := simpl; try (move: addSn addnS; congruence).
```

A.2 Untyped.v

```
Require Import
  Coq.Relations.Relations Coq.Relations.Relation_Operators
  Ssreflect.ssreflect Ssreflect.ssrfun Ssreflect.ssrbool Ssreflect.eqtype
  Ssreflect.ssrnat Ssreflect.seq
  LCAC.lib.Relations_ext LCAC.lib.seq_ext_base LCAC.lib.ssrnat_ext
  LCAC.lib.seq_ext.

Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

A.2.1 項の定義

```
Inductive term : Set := var of nat | app of term & term | abs of term.

Coercion var : nat >-> term.

Fixpoint eqterm t1 t2 :=
  match t1, t2 with
    | var n, var m => n == m
    | app t1l t1r, app t2l t2r => eqterm t1l t2l && eqterm t1r t2r
    | abs t1, abs t2 => eqterm t1 t2
    | _, _ => false
  end.

Lemma eqtermP : Equality.axiom eqterm.
Proof.
  move => t1 t2; apply: (iffP idP) => [| <-].
  - by elim: t1 t2 => [n | t1l IH t1r IH' | t1 IH]
      [// m /eqP -> | //= t2l t2r /andP [] /IH -> /IH' -> | // t2 /IH ->].
  - by elim: t1 => //= t1l ->.
Defined.

Canonical term_eqMixin := EqMixin eqtermP.
Canonical term_eqType := Eval hnf in EqType term term_eqMixin.
```

A.2.2 シフトと代入の定義

```
Fixpoint shift d c t : term :=
  match t with
    | var n => var (if c <= n then n + d else n)
    | app t1 t2 => app (shift d c t1) (shift d c t2)
    | abs t1 => abs (shift d c.+1 t1)
  end.

Notation substitutev ts m n :=
  (shift n 0 (nth (var (m - n - size ts)) ts (m - n))) (only parsing).
```

```
Fixpoint substitute n ts t : term :=
  match t with
    | var m => if n <= m then substitutev ts m n else m
    | app t1 t2 => app (substitute n ts t1) (substitute n ts t2)
    | abs t' => abs (substitute n.+1 ts t')
  end.
```

A.2.3　簡約の定義

```
Reserved Notation "t ->b1 t'" (at level 70, no associativity).

Inductive betared1 : relation term :=
  | betared1beta t1 t2     : app (abs t1) t2 ->b1 substitute 0 [:: t2] t1
  | betared1appl t1 t1' t2 : t1 ->b1 t1' -> app t1 t2 ->b1 app t1' t2
  | betared1appr t1 t2 t2' : t2 ->b1 t2' -> app t1 t2 ->b1 app t1 t2'
  | betared1abs t t'       : t ->b1 t' -> abs t ->b1 abs t'
  where "t ->b1 t'" := (betared1 t t').

Notation betared := [* betared1].
Infix "->b" := betared (at level 70, no associativity).

Hint Constructors betared1.
```

A.3　STLC.v

```
Require Import
  Coq.Relations.Relations Coq.Relations.Relation_Operators
  Ssreflect.ssreflect Ssreflect.ssrfun Ssreflect.ssrbool Ssreflect.eqtype
  Ssreflect.ssrnat Ssreflect.seq
  LCAC.lib.Relations_ext LCAC.lib.seq_ext_base LCAC.lib.ssrnat_ext
  LCAC.lib.seq_ext.

Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

A.3.1　型と項の定義

```
Inductive typ := tyvar of nat | tyfun of typ & typ.
Inductive term := var of nat | app of term & term | abs of typ & term.

Coercion tyvar : nat >-> typ.
Coercion var : nat >-> term.

Notation "ty :->: ty'" := (tyfun ty ty') (at level 50, no associativity).
Notation "t @ t'" := (app t t') (at level 60, no associativity).

Fixpoint eqtyp t1 t2 :=
  match t1, t2 with
```

```
    | tyvar n, tyvar m => n == m
    | t1l :->: t1r, t2l :->: t2r => eqtyp t1l t2l && eqtyp t1r t2r
    | _, _ => false
  end.

Lemma eqtypP : Equality.axiom eqtyp.
Proof.
  move => t1 t2; apply: (iffP idP) => [| <-].
  - by elim: t1 t2 => [n | t1l IHt1l t1r IHt1r]
      [// m /eqP -> | //= t2l t2r /andP [] /IHt1l -> /IHt1r ->].
  - by elim: t1 => //= t1l ->.
Defined.

Canonical typ_eqMixin := EqMixin eqtypP.
Canonical typ_eqType := Eval hnf in EqType typ typ_eqMixin.

Fixpoint eqterm t1 t2 :=
  match t1, t2 with
    | var n, var m => n == m
    | t1l @ t1r, t2l @ t2r => eqterm t1l t2l && eqterm t1r t2r
    | abs ty1 t1, abs ty2 t2 => (ty1 == ty2) && eqterm t1 t2
    | _, _ => false
  end.

Lemma eqtermP : Equality.axiom eqterm.
Proof.
  move => t1 t2; apply: (iffP idP) => [| <-].
  - by elim: t1 t2 => [n | t1l IHt1l t1r IHt1r | ty1 t1 IHt1]
      [// m /eqP -> | //= t2l t2r /andP [] /IHt1l -> /IHt1r -> |
       //= ty2 t2 /andP [] /eqP -> /IHt1 ->].
  - by elim: t1 => //= [t1l -> | ty1 t1 ->] //; rewrite andbT.
Defined.

Canonical term_eqMixin := EqMixin eqtermP.
Canonical term_eqType := Eval hnf in EqType term term_eqMixin.
```

A.3.2　シフトと代入の定義

```
Fixpoint shift d c t : term :=
  match t with
    | var n => var (if c <= n then n + d else n)
    | t1 @ t2 => shift d c t1 @ shift d c t2
    | abs ty t1 => abs ty (shift d c.+1 t1)
  end.

Notation substitutev ts m n :=
  (shift n 0 (nth (var (m - n - size ts)) ts (m - n))) (only parsing).

Fixpoint substitute n ts t : term :=
  match t with
    | var m => if n <= m then substitutev ts m n else m
    | t1 @ t2 => substitute n ts t1 @ substitute n ts t2
    | abs ty t' => abs ty (substitute n.+1 ts t')
  end.
```

A.3.3 簡約の定義

```
Reserved Notation "t ->b1 t'" (at level 80, no associativity).

Inductive betared1 : relation term :=
  | betared1beta ty t1 t2 : abs ty t1 @ t2 ->b1 substitute 0 [:: t2] t1
  | betared1appl t1 t1' t2 : t1 ->b1 t1' -> t1 @ t2 ->b1 t1' @ t2
  | betared1appr t1 t2 t2' : t2 ->b1 t2' -> t1 @ t2 ->b1 t1 @ t2'
  | betared1abs ty t t' : t ->b1 t' -> abs ty t ->b1 abs ty t'
  where "t ->b1 t'" := (betared1 t t').

Notation betared := [* betared1].
Infix "->b" := betared (at level 80, no associativity).

Hint Constructors betared1.
```

A.3.4 型付け規則の定義

```
Fixpoint typing_rec (ctx : context typ) (t : term) : option typ :=
  match t with
    | var n => nth None ctx n
    | tl @ tr =>
      if typing_rec ctx tl is Some (tyl :->: tyr)
        then (if typing_rec ctx tr == Some tyl then Some tyr else None)
        else None
    | abs ty t => omap (tyfun ty) (typing_rec (Some ty :: ctx) t)
  end.

Definition typing := nosimpl typing_rec.

Notation "ctx \|- t \: ty" := (Some ty == typing ctx t)
  (at level 69, no associativity).
Notation "ctx \|- t \: ty" := (Some (ty : typ) == typing ctx t)
  (at level 69, no associativity, only parsing).
```

A.3.5 逆転補題

```
Lemma typing_varE ctx (n : nat) ty : ctx \|- n \: ty = ctxindex ctx n ty.
Proof. by rewrite /typing /=. Qed.

Lemma typing_appP ctx t1 t2 ty :
  reflect (exists2 tyl, ctx \|- t1 \: tyl :->: ty & ctx \|- t2 \: tyl)
          (ctx \|- t1 @ t2 \: ty).
Proof.
  apply: (iffP idP); rewrite /typing /=.
  - by move: (typing_rec ctx t1) => [] // [] // tyl tyr;
      case: ifP => // /eqP -> /eqP [] ->; exists tyl.
  - by case => tyl /eqP <- /eqP <-; rewrite eqxx.
Qed.
```

```
Lemma typing_absP ctx t tyl ty :
  reflect (exists2 tyr, ty = tyl :->: tyr & Some tyl :: ctx \|- t \: tyr)
          (ctx \|- abs tyl t \: ty).
Proof.
  apply: (iffP idP); rewrite /typing /=.
  - by case: typing_rec => //= tyr /eqP [] ->; exists tyr.
  - by case => tyr ->; case: typing_rec => // tyr' /eqP [] <-.
Qed.

Lemma typing_absE ctx t tyl tyr :
  ctx \|- abs tyl t \: tyl :->: tyr = Some tyl :: ctx \|- t \: tyr.
Proof.
  by rewrite /typing /=; case: typing_rec => //= tyr';
    rewrite /eq_op /= /eq_op /= -/eq_op eqxx.
Qed.
```

A.4 F.v

```
Require Import
  Coq.Relations.Relations Coq.Relations.Relation_Operators
  Ssreflect.ssreflect Ssreflect.ssrfun Ssreflect.ssrbool Ssreflect.eqtype
  Ssreflect.ssrnat Ssreflect.seq
  LCAC.lib.Relations_ext LCAC.lib.seq_ext_base LCAC.lib.ssrnat_ext
  LCAC.lib.seq_ext.

Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

A.4.1 型と項の定義

```
Inductive typ := tyvar of nat | tyfun of typ & typ | tyabs of typ.

Inductive term
  := var  of nat                        (* variable *)
   | app  of term & term                (* value application *)
   | abs  of typ & term                 (* value abstraction *)
   | uapp of term & typ                 (* universal application *)
   | uabs of term.                      (* universal abstraction *)

Coercion tyvar : nat >-> typ.
Coercion var : nat >-> term.

Notation "ty :->: ty'" := (tyfun ty ty') (at level 50, no associativity).
Notation "t @ t'" := (app t t') (at level 60, no associativity).
Notation "t @' ty" := (uapp t ty) (at level 60, no associativity).

Fixpoint eqtyp t1 t2 :=
  match t1, t2 with
    | tyvar n, tyvar m => n == m
    | t1l :->: t1r, t2l :->: t2r => eqtyp t1l t2l && eqtyp t1r t2r
    | tyabs tl, tyabs tr => eqtyp tl tr
```

```
    | _, _ => false
  end.

Lemma eqtypP : Equality.axiom eqtyp.
Proof.
  move => t1 t2; apply: (iffP idP) => [| <-].
  - by elim: t1 t2 => [n | t1l IHt1l t1r IHt1r | t1 IHt1]
      [// m /eqP -> | //= t2l t2r /andP [] /IHt1l -> /IHt1r -> |
       // t2 /IHt1 ->].
  - by elim: t1 => //= t ->.
Defined.

Canonical typ_eqMixin := EqMixin eqtypP.
Canonical typ_eqType := Eval hnf in EqType typ typ_eqMixin.

Fixpoint eqterm t1 t2 :=
  match t1, t2 with
    | var n, var m => n == m
    | t1l @ t1r, t2l @ t2r => eqterm t1l t2l && eqterm t1r t2r
    | abs ty1 t1, abs ty2 t2 => (ty1 == ty2) && eqterm t1 t2
    | t1 @' ty1, t2 @' ty2 => eqterm t1 t2 && (ty1 == ty2)
    | uabs t1, uabs t2 => eqterm t1 t2
    | _, _ => false
  end.

Lemma eqtermP : Equality.axiom eqterm.
Proof.
  move => t1 t2; apply: (iffP idP) => [| <-].
  - by elim: t1 t2 =>
      [n | t1l IHt1l t1r IHt1r | ty1 t1 IHt1 | t1 IHt1 ty1 | t1 IHt1]
      [// m /eqP -> | //= t2l t2r /andP [] /IHt1l -> /IHt1r -> |
       //= ty2 t2 /andP [] /eqP -> /IHt1 -> |
       //= t2 ty2 /andP [] /IHt1 -> /eqP -> | //= t2 /IHt1 ->].
  - by elim: t1 => //= [t -> t' -> | ty t -> | t ->] *; rewrite ?eqxx.
Defined.

Canonical term_eqMixin := EqMixin eqtermP.
Canonical term_eqType := Eval hnf in EqType term term_eqMixin.
```

A.4.2　型のシフトと代入の定義

```
Fixpoint shift_typ d c t :=
  match t with
    | tyvar n => tyvar (if c <= n then n + d else n)
    | tl :->: tr => shift_typ d c tl :->: shift_typ d c tr
    | tyabs t => tyabs (shift_typ d c.+1 t)
  end.

Notation subst_typv ts m n :=
  (shift_typ n 0 (nth (tyvar (m - n - size ts)) ts (m - n))) (only parsing).

Fixpoint subst_typ n ts t :=
  match t with
    | tyvar m => if n <= m then subst_typv ts m n else m
```

```
    | tl :->: tr => subst_typ n ts tl :->: subst_typ n ts tr
    | tyabs t => tyabs (subst_typ n.+1 ts t)
  end.

Fixpoint typemap (f : nat -> typ -> typ) n t :=
  match t with
    | var m => var m
    | tl @ tr => typemap f n tl @ typemap f n tr
    | abs ty t => abs (f n ty) (typemap f n t)
    | t @' ty => typemap f n t @' f n ty
    | uabs t => uabs (typemap f n.+1 t)
  end.
```

A.4.3　項のシフトと代入の定義

```
Fixpoint shift_term d c t :=
  match t with
    | var n => var (if c <= n then n + d else n)
    | tl @ tr => shift_term d c tl @ shift_term d c tr
    | abs ty t => abs ty (shift_term d c.+1 t)
    | t @' ty => shift_term d c t @' ty
    | uabs t => uabs (shift_term d c t)
  end.

Notation subst_termv ts m n n' :=
  (typemap (shift_typ n') 0
    (shift_term n 0 (nth (var (m - n - size ts)) ts (m - n)))) (only parsing).

Fixpoint subst_term n n' ts t :=
  match t with
    | var m => if n <= m then subst_termv ts m n n' else m
    | tl @ tr => subst_term n n' ts tl @ subst_term n n' ts tr
    | abs ty t => abs ty (subst_term n.+1 n' ts t)
    | t @' ty => subst_term n n' ts t @' ty
    | uabs t => uabs (subst_term n n'.+1 ts t)
  end.
```

A.4.4　簡約の定義

```
Reserved Notation "t ->r1 t'" (at level 70, no associativity).

Inductive reduction1 : relation term :=
  | red1fst ty t1 t2   : abs ty t1 @ t2 ->r1 subst_term 0 0 [:: t2] t1
  | red1snd t ty       : uabs t @' ty ->r1 typemap (subst_typ^~ [:: ty]) 0 t
  | red1appl t1 t1' t2 : t1 ->r1 t1' -> t1 @ t2 ->r1 t1' @ t2
  | red1appr t1 t2 t2' : t2 ->r1 t2' -> t1 @ t2 ->r1 t1 @ t2'
  | red1abs ty t t'    : t ->r1 t' -> abs ty t ->r1 abs ty t'
  | red1uapp t t' ty   : t ->r1 t' -> t @' ty ->r1 t' @' ty
  | red1uabs t t'      : t ->r1 t' -> uabs t ->r1 uabs t'
  where "t ->r1 t'" := (reduction1 t t').

Notation reduction := [* reduction1].
```

```
Infix "->r" := reduction (at level 70, no associativity).

Hint Constructors reduction1.
```

A.4.5　型付け規則の定義

```
Fixpoint typing_rec (ctx : context typ) (t : term) : option typ :=
  match t with
    | var n => nth None ctx n
    | tl @ tr =>
      if typing_rec ctx tl is Some (tyl :->: tyr)
        then (if typing_rec ctx tr == Some tyl then Some tyr else None)
        else None
    | abs ty t => omap (tyfun ty) (typing_rec (Some ty :: ctx) t)
    | t @' ty =>
      if typing_rec ctx t is Some (tyabs ty')
        then Some (subst_typ 0 [:: ty] ty')
        else None
    | uabs t => omap tyabs (typing_rec (ctxmap (shift_typ 1 0) ctx) t)
  end.

Definition typing := nosimpl typing_rec.

Notation "ctx \|- t \: ty" := (Some ty == typing ctx t)
  (at level 69, no associativity).
Notation "ctx \|- t \: ty" := (Some (ty : typ) == typing ctx t)
  (at level 69, no associativity, only parsing).
```

A.4.6　逆転補題

```
Lemma typing_varE ctx (n : nat) (ty : typ) :
  ctx \|- n \: ty = ctxindex ctx n ty.
Proof. by rewrite /typing /=. Qed.

Lemma typing_appP ctx t1 t2 ty :
  reflect (exists2 tyl, ctx \|- t1 \: tyl :->: ty & ctx \|- t2 \: tyl)
          (ctx \|- t1 @ t2 \: ty).
Proof.
  apply: (iffP idP); rewrite /typing /=.
  - by move: (typing_rec ctx t1) => [] // [] // tyl tyr;
      case: ifP => // /eqP -> /eqP [] ->; exists tyl.
  - by case => tyl /eqP <- /eqP <-; rewrite eqxx.
Qed.

Lemma typing_absP ctx t tyl ty :
  reflect (exists2 tyr, ty = tyl :->: tyr & Some tyl :: ctx \|- t \: tyr)
          (ctx \|- abs tyl t \: ty).
Proof.
  apply: (iffP idP); rewrite /typing /=.
  - by case: typing_rec => //= tyr /eqP [] ->; exists tyr.
  - by case => tyr ->; case: typing_rec => // tyr' /eqP [] <-.
Qed.
```

```
Lemma typing_absE ctx t tyl tyr :
  ctx \|- abs tyl t \: tyl :->: tyr = Some tyl :: ctx \|- t \: tyr.
Proof.
  by rewrite /typing /=; case: typing_rec => //= tyr';
    rewrite /eq_op /= /eq_op /= -/eq_op eqxx.
Qed.

Lemma typing_absE' ctx t tyl tyl' tyr :
  ctx \|- abs tyl t \: tyl' :->: tyr =
  (tyl == tyl') && (Some tyl :: ctx \|- t \: tyr).
Proof.
  rewrite /typing /=; case: typing_rec => //=.
  - by move => tyr'; rewrite !/eq_op /= /eq_op /= -/eq_op eq_sym.
  - by rewrite andbF.
Qed.

Lemma typing_uappP ctx t ty1 ty2 :
  reflect
    (exists2 ty3, ty2 = subst_typ 0 [:: ty1] ty3 & ctx \|- t \: tyabs ty3)
    (ctx \|- t @' ty1 \: ty2).
Proof.
  apply: (iffP idP); rewrite /typing /=.
  - by move: (typing_rec ctx t) => [] // [] // ty3 /eqP [->]; exists ty3.
  - by case => t3 -> /eqP <-.
Qed.

Lemma typing_uabsP ctx t ty :
  reflect
    (exists2 ty', ty = tyabs ty' & ctxmap (shift_typ 1 0) ctx \|- t \: ty')
    (ctx \|- uabs t \: ty).
Proof.
  apply: (iffP idP); rewrite /typing /=.
  - by case: typing_rec => // ty' /eqP [] ->; exists ty'.
  - by case => ty' -> /eqP; case: typing_rec => // ty'' <-.
Qed.

Lemma typing_uabsE ctx t ty :
  ctx \|- uabs t \: tyabs ty = ctxmap (shift_typ 1 0) ctx \|- t \: ty.
Proof. by rewrite /typing /=; case: typing_rec. Qed.
```

付録 B

証明

B.1 補題 2.22

Proof. 右から左は SN-Intro と一致するので明らか．左から右は，集合 $P \subseteq A$ を

$$P(x) = (\forall y \in A.\, x \rightsquigarrow y \Rightarrow \mathrm{SN}_{\rightsquigarrow}(y))$$

と定義すると以下のように証明できる．

$$\begin{aligned}
& \forall x \in A.\, P(x) \Rightarrow P(x) && \text{(自明な恒真命題)} \\
\Rightarrow\ & \forall x \in A.\, (\forall y \in A.\, x \rightsquigarrow y \Rightarrow \mathrm{SN}_{\rightsquigarrow}(y)) \Rightarrow P(x) && (P \text{ を展開}) \\
\Rightarrow\ & \forall x \in A.\, \mathrm{SN}_{\rightsquigarrow}(x) \wedge (\forall y \in A.\, x \rightsquigarrow y \Rightarrow \mathrm{SN}_{\rightsquigarrow}(y) \wedge P(y)) \\
& \quad \Rightarrow P(x) && \text{(仮定を強める)} \\
\Rightarrow\ & \mathrm{SN}_{\rightsquigarrow} \subseteq P && \text{(SN-Elim を適用)} \\
\Rightarrow\ & \forall x \in A.\, \mathrm{SN}_{\rightsquigarrow}(x) \Rightarrow (\forall y \in A.\, x \rightsquigarrow y \Rightarrow \mathrm{SN}_{\rightsquigarrow}(y)) && (\subseteq \text{ と } P \text{ を展開}) \quad \square
\end{aligned}$$

B.2 補題 2.23

Proof.

$$\begin{aligned}
& \forall x \in A.\, (\forall y \in A.\, x \rightsquigarrow y \Rightarrow P(y)) \Rightarrow P(x) \\
\Rightarrow\ & \forall x \in A.\, \mathrm{SN}_{\rightsquigarrow}(x) \wedge (\forall y \in A.\, x \rightsquigarrow y \Rightarrow \mathrm{SN}_{\rightsquigarrow}(y) \wedge P(y)) \\
& \quad \Rightarrow P(x) && \text{(仮定を強める)} \\
\Rightarrow\ & \mathrm{SN}_{\rightsquigarrow} \subseteq P && \text{(SN-Elim を適用)} \quad \square
\end{aligned}$$

B.3 補題 2.24

Proof. 集合 $P \subseteq B$ を

$$P(x') = (\forall x \in A.\, x' = f(x) \Rightarrow \mathrm{SN}_{\rightsquigarrow_A}(x))$$

と定義する．この P について，

$$\forall x \in A.\, (\forall y' \in B.\, f(x) \rightsquigarrow_B y' \Rightarrow P(y')) \Rightarrow \mathrm{SN}_{\rightsquigarrow_A}(x)$$

を証明する．

$$
\begin{aligned}
&\forall y' \in B.\, f(x) \rightsquigarrow_B y' \Rightarrow P(y') \\
\Rightarrow\ &\forall y' \in B.\, f(x) \rightsquigarrow_B y' \Rightarrow (\forall y \in A.\, y' = f(y) \Rightarrow \mathrm{SN}_{\rightsquigarrow_A}(y)) && (P \text{ を展開}) \\
\Rightarrow\ &\forall y \in A, y' \in B.\, y' = f(y) \wedge f(x) \rightsquigarrow_B y' \Rightarrow \mathrm{SN}_{\rightsquigarrow_A}(y) \\
\Rightarrow\ &\forall y \in A.\, f(x) \rightsquigarrow_B f(y) \Rightarrow \mathrm{SN}_{\rightsquigarrow_A}(y) && (y' \text{ を代入}) \\
\Rightarrow\ &\forall y \in A.\, x \rightsquigarrow_A y \Rightarrow \mathrm{SN}_{\rightsquigarrow_A}(y) && (x \rightsquigarrow_A y \Rightarrow f(x) \rightsquigarrow_B f(y)) \\
\Rightarrow\ &\mathrm{SN}_{\rightsquigarrow_A}(x) && (\text{補題 2.22})
\end{aligned}
$$

直前で証明した命題より，残りの証明を以下のように書ける．

$$
\begin{aligned}
&\forall x \in A.\, (\forall y' \in B.\, f(x) \rightsquigarrow_B y' \Rightarrow P(y')) \Rightarrow \mathrm{SN}_{\rightsquigarrow_A}(x) \\
\Rightarrow\ &\forall x' \in B.\, (\forall y' \in B.\, x' \rightsquigarrow_B y' \Rightarrow P(y')) \\
&\qquad \Rightarrow (\forall x \in A.\, x' = f(x) \Rightarrow \mathrm{SN}_{\rightsquigarrow_A}(x)) && (x' = f(x)) \\
\Rightarrow\ &\forall x' \in B.\, (\forall y' \in B.\, x' \rightsquigarrow_B y' \Rightarrow P(y')) \Rightarrow P(x') && (P \text{ を畳み込み}) \\
\Rightarrow\ &\mathrm{SN}_{\rightsquigarrow_B} \subseteq P && (\text{SN-Elim}') \\
\Rightarrow\ &\forall x' \in B.\, \mathrm{SN}_{\rightsquigarrow_B}(x') \Rightarrow P(x') \\
\Rightarrow\ &\forall x' \in B.\, \mathrm{SN}_{\rightsquigarrow_B}(x') \Rightarrow (\forall x \in A.\, x' = f(x) \Rightarrow \mathrm{SN}_{\rightsquigarrow_A}(x)) && (P \text{ を展開}) \\
\Rightarrow\ &\forall x \in A, x' \in B.\, x' = f(x) \wedge \mathrm{SN}_{\rightsquigarrow_B}(x') \Rightarrow \mathrm{SN}_{\rightsquigarrow_A}(x) \\
\Rightarrow\ &\forall x \in A.\, \mathrm{SN}_{\rightsquigarrow_B}(f(x)) \Rightarrow \mathrm{SN}_{\rightsquigarrow_A}(x) && (x' \text{ を代入})
\end{aligned}
$$

□