並列代入に対する代入補題の自動証明

坂口 和彦

筑波大学 情報学群 情報科学類
sakaguchi@coins.tsukuba.ac.jp

概 要　代入補題は λ 計算の代入についての基本的な性質であり，Church-Rosser の定理や強正規化定理などの証明にも使われる重要な補題である．本研究では，de Bruijn 表現の λ 項と並列代入についての代入補題を，定理証明器 Coq で半自動的に証明する方法を提案する．De Bruijn 表現は形式的な証明に向いた λ 項やプログラムの表現として知られているが，その表現の上での代入補題の証明には多くの補題と場合分けを要する．本研究の提案手法では，この場合分けのうち自明なものを自動で証明するため，Coq に実装された等式のソルバ`congruence`とプレスバーガー算術のソルバ`lia`を用いる．また，この自動証明は代入補題の証明以外にも有用である．その適用例として System F の強正規化定理の形式的証明を紹介する．

1　はじめに

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「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」が知られている．我々は定理証明器 Coq [14] とその拡張 SSReflect [6] 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 [10] に取り組んでおり，本論文ではその証明のための道具として考案した並列代入に対する代入補題の自動証明法について説明する．

並列代入 (*parallel substitution*) とは，項中の複数の自由変数に同時に置き換えられるように定義した代入のことである．本論文中で扱っている λ 計算の体系を定義する上では並列代入を直接使うことは無いが，並列代入は強正規化定理の証明に有用である．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，代入の適用順序の入れ替えに関する補題である．代入補題を含む λ 計算に関する多くの性質は，変数名を使う通常の λ 項の表現を使って厳密に証明しようと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) が必要であり煩雑な証明になる．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** (*de Bruijn representation, nameless representation*) [2] 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 [9]．しかし，これらの定義を並列代入に拡張した場合に限り，元々証明に向かないとされた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．

代入補題の自動証明の重要な部分は，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`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` [1] 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` [14, 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．

代入補題の自動証明の仕組みは，本来の目的であるところの強正規化定理の証明にも応用できる．本論文では，我々が提案する自動証明法が System F の強正規化定理の証明にどのように応用できるかについても説明する．

2 De Bruijn 表現と並列代入

De Bruijn 表現 [2] では，λ 計算における λ 抽象などの変数束縛の位置には変数に関する情報を書かず，被束縛の位置に束縛を指す番号を持つことで項を表現する．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．

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

この項の集合の定義は，Coq では以下のように書ける．

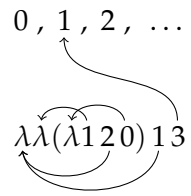
```
Inductive term : Set := var of nat | app of term & term | abs of term.

Coercion var : nat >-> term.
```

通常の λ 項の表記と同様に，以下の規則に従って括弧を省略しても良いものとする．

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

De Bruijn 表現での束縛と被束縛の対応は，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．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．



u 中の自由変数 x を t で置き換えた項を $u[x := t]$ と表記し，この置き換えの操作を代入と呼ぶ．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 [1]．
- 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 2 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 2 のように定義できる [2]．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．

[1] 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．
[2] 図 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．

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

図 1. 型無し λ 計算のシフトの定義

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

図 2. 型無し λ 計算の代入の定義 (1)

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

図 3. 型無し λ 計算の代入の定義 (2)

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 [3] 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 1 で定義されるシフト (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 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，代入位置とシフトの量がどちらも 1 ずつ増えていくことを利用して最後に t を使うときに代入位置の分だけシフトするようにしたのが図 3 の定義である．この定義は Huet [7] 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 2 の定義とは異なる場合があるが，詳しくは後程説明する．

以上の 2 つの代入の定義を並列代入に拡張すると，図 4 と図 5 の定義が得られる．この 2 つの定義における $\overline{t}$ は項の列であり，$\overline{t} \uparrow^d$ は $[t \uparrow^d |\, t \leftarrow \overline{t}]$ の略記である．$u\{x := \overline{t}\}$ と $u[x := \overline{t}]$ はどちらも直感的には u 中の自由変数 $x, \ldots, x + |\overline{t}| - 1$ に項 $t_0, \ldots, t_{|\overline{t}|-1}$ を代入した項である．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，$x \leq y < x + |\overline{t}|$ の場合は変数 y 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

[3] 本論文中での自然数の減算は，$n \leq m$ のときに $n - m = 0$ となるように定義されているものとする．

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

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

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

図 4. 型無し λ 計算の並列代入の定義 (1)

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

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

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

図 5. 型無し λ 計算の並列代入の定義 (2)

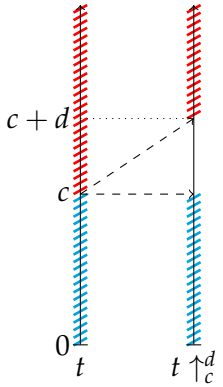


図 6. シフトによる自由変数の並びの変化

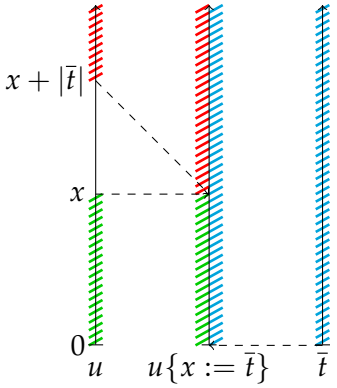


図 7. 図 4 の代入による自由変数の並びの変化

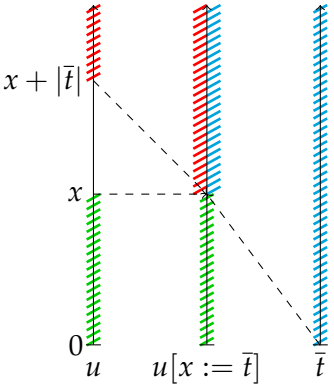


図 8. 図 5 の代入による自由変数の並びの変化

$\bar{t}_{y-x}$(もしくは $\bar{t}_{y-x}\uparrow^x$) となる．残りの場合は $y < x$ と $x + |\bar{t}| \le y$ 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．

図 2 と図 3 で定義される代入がそれぞれ図 4 と図 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 7 と図 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 1 では $t\uparrow^d$ 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 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 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．

Coq でのシフト (`shift`) と代入 (`substitute`) の定義を以下に示す．ここでの代入の定義は図 5 の定義に対応する．元の代入の定義では項が変数の場合について 3 つの場合分けをしているが，以下では $x + |\bar{t}| \le y$ の場合と $x \le y < x + |\bar{t}|$ の場合はまとめて $\mathrm{nth}(y - x - |\bar{t}|, \bar{t}, y - x)\uparrow_0^x$ としている．ただし，ここでの $\mathrm{nth}(d, \bar{s}, n)$ は，$n < |\bar{s}|$ の場合は $\bar{s}_n$，それ以外の場合は d である．

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

図 9. β 簡約の定義

$$t \uparrow_n^0 = t \tag{1}$$
$$c \le c' \le c + d \Rightarrow t \uparrow_c^d \uparrow_{c'}^{d'} = t \uparrow_c^{d'+d} \tag{2}$$
$$c' \le c \Rightarrow t \uparrow_c^d \uparrow_{c'}^{d'} = t \uparrow_{c'}^{d'} \uparrow_{d'+c}^{d} \tag{3}$$
$$c \le n \Rightarrow t[n := \overline{u}] \uparrow_c^d = t \uparrow_c^d [d + n := \overline{u}] \tag{4}$$
$$n \le c \Rightarrow t[n := \overline{u}] \uparrow_c^d = t \uparrow_{|\overline{u}|+c}^{d} [n := \overline{u} \uparrow_{c-n}^{d}] \tag{5}$$
$$c \le n \wedge |\overline{u}| + n \le d + c \Rightarrow t \uparrow_c^d [n := \overline{u}] = t \uparrow_c^{d-|\overline{u}|} \tag{6}$$
$$m \le n \Rightarrow t[m := \overline{u}][n := \overline{v}] = t[|\overline{u}| + n := \overline{v}][m := \overline{u}[n - m := \overline{v}]] \tag{7}$$
$$t[|\overline{v}| + n := \overline{u}][n := \overline{v}] = t[n := \overline{v} +\!\!+ \overline{u}] \tag{8}$$
$$t[n := []] = t \tag{9}$$

図 10. 型無し λ 計算のシフトと代入の性質

```
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))).

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.
```

3　代入補題の半自動証明

本節では，代入補題だけではなく図 10 に示すようなシフトと代入についての多くの代数的性質を Coq 上で自動的に証明する方法を検討する．ここでの $\overline{u}[x := t]$ は $[u[x := t] \mid u \leftarrow \overline{u}]$ の略記である．これらの補題の証明間の依存関係を図 11 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 (3), (4), (5), (7) のような補題であり，特に補題 (7) は代入と代入の適用順序を入れ替えるためのものであるため，代入補題に相当する．

これら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

$$\forall m.\, n \le m \Rightarrow P[n, m]$$
$$\Leftrightarrow \forall m'.\, P[n, n + m']$$

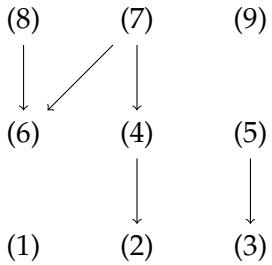


図 11. 図 10 の補題の証明間の依存関係

という性質を用いて除去できる．例えば，補題 (5) はこの性質を使うことで

$$t[n := \overline{u}] \uparrow^d_{n+c} = t \uparrow^d_{|\overline{u}|+(n+c)} [n := \overline{u} \uparrow^d_c] \tag{10}$$

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 $((\lambda\, t_1 \uparrow^d_{c+1})\, t_2 \uparrow^d_c)$ を 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 (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 (10) の左辺の形に一致させる必要がある．補題`add0n` : $\forall n.\, 0 + n = n$ 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 (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 10 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` : $\forall m\, n.\, \mathsf{S}\, m + n = \mathsf{S}(m + n)$
- `addnS` : $\forall m\, n.\, m + \mathsf{S}\, n = \mathsf{S}(m + n)$

による書き換えだけで証明できる．例として補題 (5) の証明の一部を見る．

Proof. (5) の仮定を上述の方法で除去して得られる命題 (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

   (省略)

2. $t = t_1\, t_2$ のとき

$$
\begin{aligned}
&(t_1\, t_2)[n := \overline{u}] \uparrow^d_{n+c} \\
&= (t_1[n := \overline{u}] \uparrow^d_{n+c})\, (t_2[n := \overline{u}] \uparrow^d_{n+c}) && \text{(定義の展開)} \\
&= (t_1 \uparrow^d_{|\overline{u}|+(n+c)} [n := \overline{u} \uparrow^d_c])\, (t_2[n := \overline{u}] \uparrow^d_{n+c}) && (t_1 \text{ についての帰納法の仮定}) \\
&= (t_1 \uparrow^d_{|\overline{u}|+(n+c)} [n := \overline{u} \uparrow^d_c])\, (t_2 \uparrow^d_{|\overline{u}|+(n+c)} [n := \overline{u} \uparrow^d_c]) && (t_2 \text{ についての帰納法の仮定}) \\
&= (t_1\, t_2) \uparrow^d_{|\overline{u}|+(n+c)} [n := \overline{u} \uparrow^d_c] && \text{(定義の畳み込み)}
\end{aligned}
$$

3. $t = \lambda\, t_1$ のとき

$$
\begin{aligned}
&(\lambda\, t_1)[n := \overline{u}] \uparrow^d_{n+c} \\
&= \lambda\, t_1[\mathrm{S}\, n := \overline{u}] \uparrow^d_{\mathrm{S}(n+c)} && \text{(定義の展開)} \\
&= \lambda\, t_1[\mathrm{S}\, n := \overline{u}] \uparrow^d_{\mathrm{S}\, n+c} && \text{(補題 addSn)} \\
&= \lambda\, t_1 \uparrow^d_{|\overline{u}|+(\mathrm{S}\, n+c)} [\mathrm{S}\, n := \overline{u} \uparrow^d_c] && (t_1 \text{ についての帰納法の仮定}) \\
&= \lambda\, t_1 \uparrow^d_{|\overline{u}|+\mathrm{S}(n+c)} [\mathrm{S}\, n := \overline{u} \uparrow^d_c] && \text{(補題 addSn)} \\
&= \lambda\, t_1 \uparrow^d_{\mathrm{S}(|\overline{u}|+(n+c))} [\mathrm{S}\, n := \overline{u} \uparrow^d_c] && \text{(補題 addnS)} \\
&= (\lambda\, t_1) \uparrow^d_{|\overline{u}|+(n+c)} [n := \overline{u} \uparrow^d_c] && \text{(定義の畳み込み)}
\end{aligned}
$$

□

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 [8] に基く決定手続きなので，量化子を含まない範囲の等式の問題は必ず解けるようになっている．それに加えて，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，自然数の加算の可換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 [4]．

```
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.
```

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' := move => /=; try (move: addSn addnS; congruence).
```

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 (5) の場合については

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

[4] 長くなってしまうので省略するが，同様の方法で乗算の可換性も証明できることを確認した．また，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 と addSn は自然数の足し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．

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 [5]．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 (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」[6] であり，以下のようにして使う．

```
Require Import Omega.

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 x, y のうちの小さい方と同じ数になるので，補題 minnA は与えられた 2 つの自然数のうち小さい方を返す関数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．

[5] $n \leq x \wedge n \leq x$ 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．

[6] 厳密には，lia は完全な決定手続きになっている一方で，omega は omega nightmare と呼ばれる一部の命題を解けない [14, Section 21.5]．ただし，この差が問題になることはほとんど無い．

```
Goal forall (a b c d : nat),
  minn (minn a b) (minn c d) = minn (minn d c) (minn b a).
Proof.
  intros. 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. 仮定の除去と算術式を簡単にする操作を交互に適用
2. ゴールが自然数間の等式でなければ`f_equal` [7] 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. $n - (n - m)$, $n + (m - n)$ の形の式と`maxn`, `minn`[8] についての場合分け
4. 直前の場合分けで増えた仮定を本節の冒頭で示した方法で除去
5. ゴールを `lia` タクティクが扱える形に変形する [9]
6. `lia`タクティクの実行

1 の「算術式を簡単にする操作」は減算や不等式の両辺の加算の並びに共通して出現する項の消去であり，その結果として式が $0 \leq n$, $0 - n$, $n - 0$ などの形になった場合はそれぞれ true, 0, n に書き換える．これによって命題中の減算の数を減らしている．

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

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 10 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 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．

4 複数の代入を持つ体系の定義

本節では，System F [4, 5] の強正規化定理を形式化するための準備をする．System F は多相型付きの λ 計算であり，その定義を書くためには型と値それぞれについての代入が必要である [10]．De

[7] $f\, x_1 \ldots x_n = f\, y_1 \ldots y_n$ 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．

[8] 引数に取った 2 つの自然数のうち大きい方や小さい方を返す関数．ここで指しているのは上で示した`Notation`による定義ではなく，SSReflect のライブラリ `ssrnat.v` での定義である．

[9] 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．

[10] 一方，単純型付き λ 計算では型変数への代入は不要．

表 1. 図 10 の補題の Coq 上での行数

補題	命題の行数	証明の行数	合計
(1)	1	1	2
(2)	2	1	3
(3)	2	1	3
(4)	2	4	6
(5)	4	6	10
(6)	3	4	7
(7)	4	6	10
(8)	2	4	6
(9)	1	1	2
合計	21	28	49

$$U ::= X \in \mathbb{N} \mid (U \to U) \mid (\Pi\, U)$$

図 12. System F の型の定義

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

図 13. System F の項の定義

Bruijn 表現での System F の型と項の定義を図 12, 13 に，シフトと代入の定義を図 14, 15, 16, 17 に，型付け規則と簡約規則の定義を図 18, 19 に示す [11].

これ以降では，図 12 の $U_1 \to U_2$ を (U_1 から U_2 への) 関数型，$\Pi\, U$ を多相型と呼び，図 13 の $t\, U$ を型適用，$\Lambda\, t$ を型抽象と呼ぶ．

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 17 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 ($x \le y < x + |\overline{t}|$) 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 [12]．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 19 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 14 と図 15 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

$$\frac{\Gamma \vdash t : U \qquad X \notin \Gamma}{\Gamma \vdash \Lambda X.\, t : \Pi X.\, U}$$

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．

これらの定義をそのまま 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 14 と図 15 の下半分はどちらも似たような形をしており，実際にこの共通部分を図 20 の定義として括り出せる．この定義を使うと，$t \Uparrow_C^D$ と $t[X := \overline{U}]$ はそれぞれ $[U \Uparrow_Y^D |\ U \leftarrow t]_{Y=C}$, $[V[Y := \overline{U}] \mid U \leftarrow t]_{Y=X}$ と書ける．

[11] 図 18 の型環境 Γ は通常の型付き λ 計算の定義とは違い，型の列である．

[12] 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．

$$X \Uparrow_C^D = \begin{cases} X + d & \text{if } c \le X \\ X & \text{if } X < c \end{cases}$$

$$(U \to V) \Uparrow_C^D = (U \Uparrow_C^D) \to (V \Uparrow_C^D)$$

$$(\Pi\, U) \Uparrow_C^D = \Pi\, (U \Uparrow_{C+1}^D)$$

$$x \Uparrow_C^D = x$$

$$(t\, u) \Uparrow_C^D = (t \Uparrow_C^D)\, (u \Uparrow_C^D)$$

$$(\lambda : U.\, t) \Uparrow_C^D = \lambda : (U \Uparrow_C^D).\, (t \Uparrow_C^D)$$

$$(t\, V) \Uparrow_C^D = (t \Uparrow_C^D)\, (V \Uparrow_C^D)$$

$$(\Lambda\, t) \Uparrow_C^D = \Lambda\, (t \Uparrow_{C+1}^D)$$

図 14. 型変数のシフトの定義

$$Y[X := \overline{U}] = \begin{cases} Y - |\overline{U}| & \text{if } X + |\overline{U}| \le Y \\ \overline{U}_{Y-X} \Uparrow^X & \text{if } X \le Y < X + |\overline{U}| \\ Y & \text{if } Y < X \end{cases}$$

$$(V \to W)[X := \overline{U}] = (V[X := \overline{U}]) \to (W[X := \overline{U}])$$

$$(\Pi\, V)[X := \overline{U}] = \Pi\, (V[X + 1 := \overline{U}])$$

$$x[X := \overline{U}] = x$$

$$(t\, u)[X := \overline{U}] = (t[X := \overline{U}])\, (u[X := \overline{U}])$$

$$(\lambda : U.\, t)[X := \overline{U}] = \lambda : (U[X := \overline{U}]).\, (t[X := \overline{U}])$$

$$(t\, V)[X := \overline{U}] = (t[X := \overline{U}])\, (V[X := \overline{U}])$$

$$(\Lambda\, t)[X := \overline{U}] = \Lambda\, (t[X + 1 := \overline{U}])$$

図 15. 型変数への代入の定義

$$x \uparrow_c^d = \begin{cases} x + d & \text{if } c \le x \\ x & \text{if } x < c \end{cases}$$

$$(t\, u) \uparrow_c^d = (t \uparrow_c^d)\, (u \uparrow_c^d)$$

$$(\lambda : U.\, t) \uparrow_c^d = \lambda : U.\, (t \uparrow_{c+1}^d)$$

$$(t\, V) \uparrow_c^d = (t \uparrow_c^d)\, V$$

$$(\Lambda\, t) \uparrow_c^d = \Lambda\, (t \uparrow_c^d)$$

図 16. 変数のシフトの定義

$$y[X, x := \overline{t}] = \begin{cases} y - |\overline{U}| & \text{if } x + |\overline{t}| \le y \\ \overline{t}_{y-x} \uparrow^x \Uparrow^X & \text{if } x \le y < x + |\overline{t}| \\ y & \text{if } y < x \end{cases}$$

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

$$(\lambda : U.\, u)[X, x := \overline{t}] = \lambda : U.\, (u[X, x + 1 := \overline{t}])$$

$$(u\, V)[X, x := \overline{t}] = (u[X, x := \overline{t}])\, V$$

$$(\Lambda\, u)[X, x := \overline{t}] = \Lambda\, (u[X + 1, x := \overline{t}])$$

図 17. 変数への代入の定義

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

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

図 18. System F の型付け規則

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

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

図 19. System F の簡約規則

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

図 20. 項中の型へのマップの定義

5　強正規化定理の形式的証明

我々は，前節で述べた方法を用いて単純型付き λ 計算と System F の定義を Coq 上で記述し，それらの体系の強正規化定理をそれぞれ Tait の方法 [13] と Girard の方法 [3, 4, 5] に基いて形式化した．本節では，System F の強正規化定理の形式的証明の流れを示した後に，その証明の一部についての de Bruijn 表現特有の問題と，第 3 節で紹介した自動証明の仕組みがどのように役立つかについて説明する．強正規化定理は，以下のような定理である．

定理 1 (System F の強正規化定理) $\Gamma \vdash t : U$ であれば，t は $\rightsquigarrow$ について強正規化可能である．

項 t_1 が簡約の二項関係 $\rightsquigarrow$ について**強正規化可能** (*strongly normalizable*) であるとは，t_1 から始まる任意の簡約列 $t_1 \rightsquigarrow t_2 \rightsquigarrow \ldots$ が必ず有限長になるということである．このことを $\mathrm{SN}_{\rightsquigarrow}(t_1)$ 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．Accの定義を以下に示す [13]．

```
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.
```

強正規化性の基本的な性質として，以下が知られている．

補題 2 任意の二項関係 $R \subseteq A \times A$, $R' \subseteq B \times B$，関数 $f : A \to B$ について以下が成り立つ．

$$(\forall x, y \in A.\, x\ R\ y \Rightarrow f(x)\ R'\ f(y)) \Rightarrow (\forall x \in A.\, \mathrm{SN}_{R'}(f(x)) \Rightarrow \mathrm{SN}_R(x))$$

これ以降では System F の簡約関係 $\rightsquigarrow$ についての強正規化性のみを考えるので，$\mathrm{SN}_{\rightsquigarrow}$ を単に SN と表記する．強正規化定理は，そのままの形で型や項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．そこで，以下で定義する reducibility と呼ばれる性質を経由して証明を行う．

定義 3 (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．ただし，CR_3 で使われている $\mathrm{neutral}(t)$ は t が $\lambda : U.\,t$ か $\Lambda\,t$ の形の場合には偽であり，それ以外の場合には真である．

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

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

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

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，この reducibility candidate の定義における項は，図 13 の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 [14]．Reducibility candidate は項の集合の性質を使って定義された「reducibility の候補」だが，以下ではその具体例を作る．

[13] Acc と SN では関係の向きが逆になっていることに注意．

[14] 一方 Girard の証明では「型 U の reducibility candidate $\mathcal{R}$」という形で何の型を持つ項の集合かを明確にしているが，そのような制限を入れない方が証明を簡単に済ませられる．

定義 4 (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．

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

$$\mathrm{RED}_{U\to V}[\overline{\mathcal{R}}](t) \overset{\text{def}}{\iff} \forall u.\, \mathrm{RED}_U[\overline{\mathcal{R}}](u) \Rightarrow \mathrm{RED}_V[\overline{\mathcal{R}}](t\,u)$$

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

補題 5 SN は reducibility candidate である．

補題 6 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．

これ以降では，適当な reducibility candidate の列 $\overline{\mathcal{R}}$ と型 U 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 6 と定義 3 の CR_1 より，reducible な項は強正規化可能であることが分かる．強正規化定理の証明の残りの部分では，以下の形で型が付く項が reducible であることを証明する．

定理 7 (Reducibility) 項の列 $\bar{t}$，型環境 Γ，型の列 $\overline{U}$，reducibility candidate の列 $\overline{\mathcal{R}}$ について，$|\bar{t}| = |\Gamma| = n$, $|\overline{U}| = |\overline{\mathcal{R}}|$ とする．このとき，

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

が成り立つ．

定理 7 の $\overline{U}$ と $\overline{\mathcal{R}}$ に空列 $[]$ を入れ，$\bar{t}$ を Γ と同じ長さの 0 の並びとすると，

$$\Gamma \vdash u : U \wedge (\forall i < |\Gamma|.\, \mathrm{RED}_{\Gamma_i}[](0)) \Rightarrow \mathrm{RED}_V[](u[0 := []][0, 0 := \underbrace{0, \ldots, 0}_{|\Gamma|}])$$

が得られる．項 0 は neutral かつ正規形なので，CR_3 より $\forall i < n.\, \mathrm{RED}_{\Gamma_i}[](0)$ の部分は真である．このことと reducible な項は強正規化可能であることから，$\Gamma \vdash u : U$ ならば $u[0 := []][0, 0 := \underbrace{0, \ldots, 0}_{|\Gamma|}]$ は強正規化可能である．型の代入と値の代入がどちらも簡約を保存することは容易に示せるので，補題 2 より u も強正規化可能である [15]．よって，強正規化定理が成り立つ．

定理 7 は項 t に関する帰納法で証明するが，Girard の証明では項の形が $t\,u$, $\Lambda\,t$, $t\,U$ の場合についての補題を 1 つずつと，それ以外にもう 1 つの補題 [5, 14.2.1 Substitution] を証明している．ここでは，後者の形式的証明について検討する．

まずは準備として，定義 4 を Girard の証明での定義に近い形で以下のように再定義する．この定義での項は名前による表現になっているため，reducibility candidate の列 $\overline{\mathcal{R}}$ がそれぞれどの自由変数に対応するかを明示的に書いている．

定義 8 (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．

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

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

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

[15] 既存の多くの証明では，$\bar{t}$ を恒等代入として代入の結果が u になることを利用して証明している．それに合わせるのであれば，$\bar{t} = [0, \ldots, |\Gamma| - 1]$ として，$u[0 := []][0, 0 := 0, \ldots, |\Gamma| - 1] = u$ であることを示せば良い．

Reducibility with parameters の定義は型の代入に近い形をしているが，この操作を型の代入と合成できるというのが問題の補題である．

補題 9

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

が成り立つ．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 が変数の場合については $U = \overline{Y}_i$ と $U = \overline{X}_i (\notin \overline{Y})$ で場合分けする必要があることに注意すれば明らかであり，$U = U_1 \to U_2$ の場合も自明なので，ここでは $U = \Pi Z.\, U'$ の場合だけを示す．

Z が U' 以外に自由出現しないことに注意すると，以下のように証明できる．

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

この証明では，Z が $\overline{V}$ 中に出現しないことを利用して $\mathrm{RED}_{\overline{V}}[Z, \overline{X} := \mathcal{S}, \overline{\mathcal{R}}]$ を $\mathrm{RED}_{\overline{V}}[\overline{X} := \overline{\mathcal{R}}]$ に書き換えている．一方，de Bruijn 表現を使うのであれば明示的に列から削らないと不要な部分が使われてしまうので，何かしらの形で不要な部分を適切に削る必要がある．また，代入補題でそうだったように，代入の位置などがずれないように命題を注意深く記述する必要がある．

まず，代入の位置を 0 に限定した命題を書くと

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

とするのが適切である．左辺では $\overline{\mathcal{R}}$ は $U[0 := \overline{V}]$ の 0 以上の自由変数に順番に対応しているが，これは元々 U の $|\overline{V}|$ 以上の自由変数と $\overline{V}$ の 0 以上の自由変数であり，その点に注目すると右辺との整合性も取れている．次に，この命題を帰納法で証明できる形に一般化する．素直に帰納法を適用し $U = \Pi\, U'$ の場合に注目すると，帰納法の仮定として欲しい命題は

$$\mathrm{RED}_{U[1:=\overline{V}]}[\mathcal{S}, \overline{\mathcal{R}}](t) \Leftrightarrow \mathrm{RED}_U[\mathcal{S}, \mathrm{RED}_{\overline{V}}[\overline{\mathcal{R}}], \overline{\mathcal{R}}](t)$$

となる．よって，列を削る必要があった部分では $\overline{\mathcal{R}}$ の先頭から代入位置の分だけ要素を削り，$\mathrm{RED}_{\overline{V}}[\ldots]$ は $\overline{\mathcal{R}}$ の代入位置の部分に挿入すると十分に一般化された命題が得られるように見える．この命題を記述するために，以下の定義を導入する．

定義 10 (take, drop, insert) $\mathrm{take}(n, \overline{s})$, $\mathrm{drop}(n, \overline{s})$ をそれぞれ $\overline{s}$ の先頭 n 要素と，$\overline{s}$ から先頭 n 要素を取り除いた残りの部分とする．ただし，$|\overline{s}| < n$ の場合はそれぞれ $\overline{s}$ と空列とする．これらの定義を使って，insert を以下のように定義する．

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

$\mathrm{insert}(\overline{x}, \overline{y}, d, n)$ は，直感的には $\overline{y}$ の n 番目の位置に $\overline{x}$ を挿入して得られる列である．ただし，$\overline{y}$ の長さが n に満たない場合にも $\overline{x}$ が n 番目から始まるようにするために，結果の列の $|\overline{y}|$ 番目から $n-1$ 番目までは d で埋められる．

nth と insert について以下が成り立つ．

補題 11

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

上の定義を用いて補題 9 を de Bruijn 表現向けに書き直すと，以下のようになる．

補題 12 $X \le |\overline{\mathcal{R}}| \Rightarrow \mathrm{RED}_{U[X:=\overline{V}]}[\overline{\mathcal{R}}](t) \Leftrightarrow \mathrm{RED}_U[\mathrm{insert}(\mathrm{RED}_{\overline{V}}[\mathrm{drop}(X, \overline{\mathcal{R}})], \overline{\mathcal{R}}, \mathrm{SN}, X)](t)$

補題 12 の証明には以下の補題が必要である．

補題 13 $C \le |\overline{\mathcal{R}}| \Rightarrow \mathrm{RED}_{U \Uparrow_C^{|\mathcal{S}|}}[\mathrm{insert}(\overline{\mathcal{S}}, \overline{\mathcal{R}}, \mathrm{SN}, C)](t) \Leftrightarrow \mathrm{RED}_U[\overline{\mathcal{R}}](t)$

補題 9 の証明の流れに沿って補題 12 の証明を試みる．補題 13, 11 は使っても良いものとする．

Proof. U に関する帰納法で証明する．$U = U_1 \to U_2$ の場合は自明なので，ここではそれ以外の場合だけを示す．

1. $U = Y$ の場合
   命題中の補題 11 の左辺に一致するものを全て右辺に書き換え，第 3 節の自動証明を適用すると，以下の 2 つの場合が残る．

   (a) $X \le |\overline{\mathcal{R}}| \Rightarrow \mathrm{RED}_{\mathrm{nth}(Y, \overline{V}, |\overline{V}|+Y) \Uparrow^X}[\overline{\mathcal{R}}](t) \Leftrightarrow \mathrm{nth}(\mathrm{SN}, \overline{\mathcal{R}}, X+Y)(t)$

   $$\begin{aligned} & \mathrm{RED}_{\mathrm{nth}(Y, \overline{V}, |\overline{V}|+Y) \Uparrow^X}[\overline{\mathcal{R}}](t) \\ \Leftrightarrow\ & \mathrm{RED}_{Y \Uparrow^X}[\overline{\mathcal{R}}](t) && (|\overline{V}| \le |\overline{V}| + Y) \\ \Leftrightarrow\ & \mathrm{RED}_{X+Y}[\overline{\mathcal{R}}](t) && (\Uparrow \text{ の定義，} 0 \le Y) \\ \Leftrightarrow\ & \mathrm{nth}(\mathrm{SN}, \overline{\mathcal{R}}, X+Y)(t) && (\mathrm{RED} \text{ の定義}) \end{aligned}$$

   (b) $X \le |\overline{\mathcal{R}}| \wedge Y < |\overline{V}| \Rightarrow \mathrm{RED}_{\mathrm{nth}(Y-|\overline{V}|, \overline{V}, Y) \Uparrow^X}[\overline{\mathcal{R}}](t) \Leftrightarrow \mathrm{nth}(\mathrm{SN}, \mathrm{RED}_{\overline{V}}[\mathrm{drop}(X, \overline{\mathcal{R}})], Y)(t)$

   $$\begin{aligned} & \mathrm{nth}(\mathrm{SN}, \mathrm{RED}_{\overline{V}}[\mathrm{drop}(X, \overline{\mathcal{R}})], Y)(t) \\ \Leftrightarrow\ & \mathrm{RED}_{\mathrm{nth}(Y-|\overline{V}|, |\overline{V}|, Y)}[\mathrm{drop}(X, \overline{\mathcal{R}})](t) && (\mathrm{map} \text{ と } \mathrm{nth} \text{ の入れ替え}) \\ \Leftrightarrow\ & \mathrm{RED}_{\mathrm{nth}(Y-|\overline{V}|, |\overline{V}|, Y) \Uparrow^{|\mathrm{take}(X, \overline{\mathcal{R}})|}} \\ & \qquad [\mathrm{insert}(\mathrm{take}(X, \overline{\mathcal{R}}), \mathrm{drop}(X, \overline{\mathcal{R}}), \mathrm{SN}, 0)](t) && (\text{補題 13}) \\ \Leftrightarrow\ & \mathrm{RED}_{\mathrm{nth}(Y-|\overline{V}|, |\overline{V}|, Y) \Uparrow^{|\mathrm{take}(X, \overline{\mathcal{R}})|}}[\mathrm{take}(X, \overline{\mathcal{R}}), \mathrm{drop}(X, \overline{\mathcal{R}})](t) && (\mathrm{insert} \text{ の定義}) \\ \Leftrightarrow\ & \mathrm{RED}_{\mathrm{nth}(Y-|\overline{V}|, |\overline{V}|, Y) \Uparrow^{|\mathrm{take}(X, \overline{\mathcal{R}})|}}[\overline{\mathcal{R}}](t) && (\mathrm{take, drop} \text{ の性質}) \\ \Leftrightarrow\ & \mathrm{RED}_{\mathrm{nth}(Y-|\overline{V}|, |\overline{V}|, Y) \Uparrow^X}[\overline{\mathcal{R}}](t) && (\text{仮定 } X \le |\overline{\mathcal{R}}|) \end{aligned}$$

2. $U = \Pi\, U'$ の場合

   $$\begin{aligned} & \mathrm{RED}_{(\Pi\, U')[X:=\overline{V}]}[\overline{\mathcal{R}}](t) \\ \Leftrightarrow\ & \forall W, \mathcal{S}.\, \mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_{U'[X+1:=\overline{V}]}[\mathcal{S}, \overline{\mathcal{R}}](t\, W) \\ \Leftrightarrow\ & \forall W, \mathcal{S}.\, \mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_{U'}[\mathrm{insert}(\mathrm{RED}_{\overline{V}}[\mathrm{drop}(X+1, \mathcal{S} :: \overline{\mathcal{R}})], \mathcal{S} :: \overline{\mathcal{R}}, \mathrm{SN}, X+1)](t\, W) \\ \Leftrightarrow\ & \forall W, \mathcal{S}.\, \mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_{U'}[\mathrm{insert}(\mathrm{RED}_{\overline{V}}[\mathrm{drop}(X, \overline{\mathcal{R}})], \mathcal{S} :: \overline{\mathcal{R}}, \mathrm{SN}, X+1)](t\, W) \\ \Leftrightarrow\ & \forall W, \mathcal{S}.\, \mathrm{RC}(\mathcal{S}) \Rightarrow \mathrm{RED}_{U'}[\mathcal{S}, \mathrm{insert}(\mathrm{RED}_{\overline{V}}[\mathrm{drop}(X, \overline{\mathcal{R}})], \overline{\mathcal{R}}, \mathrm{SN}, X)](t\, W) \\ \Leftrightarrow\ & \mathrm{RED}_{\Pi\, U'}[\mathrm{insert}(\mathrm{RED}_{\overline{V}}[\mathrm{drop}(X, \overline{\mathcal{R}})], \overline{\mathcal{R}}, \mathrm{SN}, X)](t) \end{aligned}$$

   □

この証明を Coq で書くと以下のようになる．第 3 節の自動証明によって，4 つの場合に対する証明を省略できている．

```
Lemma subst_reducibility ty preds n tys t :
  n <= size preds ->
  (reducible (subst_typ n tys ty) preds t <->
   reducible ty
     (insert [seq reducible ty (drop n preds) | ty <- tys] preds SN n) t).
Proof.
  elim: ty preds n tys t => [v | tyl IHtyl tyr IHtyr | ty IHty] preds n tys t.
  - rewrite /= nth_insert size_map; elimif_omega.
    + by rewrite nth_default ?leq_addr //= addnC.
    + move => H0.
      rewrite (nth_map (tyvar (v - size tys))) //=.
      move: (shift_reducibility (nth (tyvar (v - size tys)) tys v)
        (take n preds) t (leq0n (size (drop n preds)))).
      by rewrite /insert take0 drop0 /= cat_take_drop size_takel.
  - by move => H /=; split => H0 t' /IHtyl => /(_ H) /H0 /IHtyr => /(_ H).
  - move => H /=; split => H0 ty' P /(H0 ty') {H0}.
    + by move/IHty => /= /(_ H); rewrite /insert /= subSS.
    + by move => H0; apply IHty.
Qed.
```

6 関連研究

Schäfer ら [11] は，de Bruijn 表現の項と並列代入 [16] 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 [12] であり，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を自動生成する仕組みなども持っている．

Nipkow [9] は，図 1 の定義を $d = 1$ の場合に制限したシフトと図 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:

$$
\begin{aligned}
c \le c' \Rightarrow t \uparrow_c^1 \uparrow_{c'+1}^1 &= t \uparrow_{c'}^1 \uparrow_c^1 \\
x \le c \Rightarrow t\{x := u\} \uparrow_c^1 &= t \uparrow_{c+1}^1 \{x := u \uparrow_c^1\} \\
c \le x \Rightarrow t\{x := u\} \uparrow_c^1 &= t \uparrow_c^1 \{x+1 := u \uparrow_c^1\} \\
t \uparrow_x^1 \{x := u\} &= t \\
x \le y \Rightarrow t\{y+1 := v \uparrow_x^1\}\{x := u\{y := v\}\} &= t\{x := u\}\{y := v\}
\end{aligned}
$$

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 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 4 の並列代入に適用しようとすると，代入補題の命題は

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

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た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 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 10 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

[16] 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．

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 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．

参考文献

[1] 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.

[2] 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.

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

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

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

[6] 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`.

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

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

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

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

[11] 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.

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

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

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