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
定理証明支援系を用いた CPS 変換の性質の形式化

主専攻　　ソフトウェアサイエンス主専攻

著者　　薄井千春

指導教員　亀山幸義 海野広志

要　旨

ラムダ計算に対するプログラム変換の 1 つである CPS 変換は、プログラム解析や検証、コントロールオペレータの導入、コンパイラでの利用など多くの応用を持ち、その性質の厳密な検証は重要である。本研究は、CPS 変換に対する型保存性と完全性について定理証明支援システム Coq を用いて形式化を行った。型保存定理は、型付けられた項を CPS 変換して得られる項は正しく型付けられるという性質であり、完全性は、CPS 変換によって項の意味が変わらないという性質である。これらはよく知られた性質であるが、後者に対する形式検証の成功例は我々の知る限り存在しない。本研究は、定理証明支援システム Coq を用いて、de Bruijn インデックスを使った表現により、型保存定理と完全性定理の鍵となる補題の証明を形式化した。

完全性定理の証明では、継続の線形性を表現する変数について特殊な取り扱いが必要であり、通常の変数と区別をした形式化を行なった。

目次

第 1 章　序論 1

第 2 章　前提知識 2
2.1　単純型付きラムダ計算 2
2.2　De Bruijn インデックスによる名無し表現の単純型付きラムダ計算 4
2.3　名前付き表現の CPS 変換と型保存定理 5
2.4　Coq を用いた名無し表現の単純型付きラムダ計算の形式化 7
2.5　コンパイラの中間表現としての CPS 変換 8

第 3 章　名無し表現の CPS 変換と型保存定理 10
3.1　名無し表現の CPS 変換とその形式化 10
3.2　Coq を用いた型保存定理の形式化 11

第 4 章　名無し表現による CPS 変換の完全性の形式化 15
4.1　完全性の証明のために 16
4.2　名前付き表現による完全性の証明 16
4.3　Coq による補題の形式化 19
4.4　逆 CPS 変換の形式化に存在する問題 21

第 5 章　関連研究 24

第 6 章　結論 25
6.1　まとめ 25
6.2　今後の課題 25

謝辞 26

参考文献 27

図目次

2.1 単純型付きラムダ計算の項の定義 2
2.2 単純型付きラムダ計算の型の定義 3
2.3 単純型付きラムダ計算における自由変数および代入の定義 3
2.4 De Bruijn インデックスによる単純型付きラムダ計算の項の定義 4
2.5 De Bruijn インデックスによる単純型付きラムダ計算のシフトと代入の定義 . . . 5
2.6 ラムダ項と値の CPS 変換の定義 6
2.7 型の CPS 変換の定義 6
2.8 同値性 6
2.9 ラムダ項と型の Coq での定義 8
2.10 型判定規則 9
2.11 シフトと代入の Coq プログラム 9

3.1 De Bruijn インデックスによる CPS 変換の定義 10
3.2 ラムダ項と値に関する CPS 変換の Coq プログラム 11
3.3 型に関する CPS 変換の Coq プログラム 12

4.1 CPS 文法の定義 16
4.2 逆 CPS 変換の定義 17
4.3 CPS 項の Coq での定義 20
4.4 改良された CPS 文法の定義（一部） 22
4.5 CPS 項に関するシフトの定義 23
4.6 CPS 項に関する代入の定義 23

第 1 章

序論

CPS 変換 [1, 2] はコンパイラやインタプリタといった言語処理系内部で、中間表現の生成方法として広く用いられているプログラム変換手法の一つである。CPS 変換によって、プログラムは継続渡し形式（Continuation Passing Style）というものに変換される。継続渡し形式のプログラムでは、関数呼び出しは末尾呼び出しとなる。CPS 変換について、健全性と完全性と呼ばれる性質が成り立つことが知られている。健全性は CPS 変換によりプログラムの意味が保存されることを示し、完全性は CPS 変換された後のプログラムが同じ意味であれば、CPS 変換する前のプログラムも同じ意味であることを表している。CPS 変換がこのような性質を持っていることは、CPS 変換を内部で用いる言語処理系への信頼性を上げることにつながる。そのような意味で健全性および完全性は重要な性質である。

CPS 変換とその性質を形式化するのにあたり、Coq という定理証明支援システムを用いて形式化を行った。Coq[3, 4] は 1989 年に初版が出た定理証明支援システムである。Coq は様々な形式証明に用いられる。定理証明支援システムは証明の記述を支援し、機械的に検証を行うことでその証明の正しさを保証することができる。証明は証明手法（以下、タクティック）を記述するスクリプト言語（以下、証明記述言語）によって記述される。形式証明の対象として考えられるのは、2014 年に相次いで話題になった OpenSSL などのソフトウェアの安全性の検証や Kepler 予想 [5]、や四色定理（2005 年に証明完了）[6] などの数学の証明などである。

CPS 変換を定理証明支援システムを用いて形式化することは、すでに様々な形でなされてきた [7, 8, 9, 10]。だが、CPS 変換の完全性は、これまで（筆者の知る限り）非形式的な証明のみが与えられていた。今回定理証明支援システム Coq を用いて、de Bruijn インデックスによる名無し表現の単純型付けラムダ計算に関する CPS 変換について、その型保存定理は証明に必要な補題を含めて、完全性に関しては重要な補題について証明を形式化した。またこの際、素朴な de Bruijn インデックスを用いた形式化では問題が発生すること、そして、変数の種類により名前空間を分割することでこの問題を回避できることを説明する。

第 2 章

前提知識

2.1 節において、非形式的な単純型付きラムダ計算を定義する[*1]。2.2 節で Coq での形式証明により則した形に、de Bruijn インデックス [13] という技法を用いて名無し表現の単純型付きラムダ計算を定義する。2.3 節では名前付き表現によって CPS 変換の定義を与え、それについて成り立つ定理を示す。2.4 節で、Coq での名無し表現の単純型付きラムダの形式化がどのように行われているかを説明する。2.5 節では、CPS 変換をコンパイラ内部での中間表現の生成手法として用いることが、どのよう利点を持つかを簡単に説明する。

2.1 単純型付きラムダ計算

単純型付きラムダ計算の式（以下、ラムダ項と呼ぶ）の集合は図 2.1 のような BNF 文法で与えられる。x は加算無限個の変数の集合 Var を代表するメタ変数である。その集合の実態は要素すべてが自然数に対応付けられた集合である。また、これ以降一般にラムダ項を e や $e', e_1, e_2, \cdots$ などと表す。

型の集合は図 2.2 のような BNF 文法で与えられる。$\tau_1 \to \tau_2$ は型 τ_1 を受け取り τ_2 を返す関数型を表している。Con は基底型であるが、具体的な何らかのラムダ項に結びつけられることを意図した型ではない。ここで、$Unit$ は CPS 変換のときにのみ導入される型であるとし、本論文で

$$
\begin{aligned}
e &:= x \\
&\mid (\lambda x.e) \\
&\mid (e_1\ e_2) \\
v &:= x \\
&\mid (\lambda x.e)
\end{aligned}
$$

図 2.1　単純型付きラムダ計算の項の定義

[*1] 単純型付けラムダ計算に関するより詳しい解説は [11] や [12] に見られる。

$$
\begin{aligned}
\tau &:= \tau_1 \to \tau_2 \\
&\mid Con \\
&\mid Unit \\
\Gamma &:= \phi \\
&\mid \Gamma, x : T
\end{aligned}
$$

図 2.2　単純型付きラムダ計算の型の定義

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda x.M) &= FV(M) \setminus \{x\} \\
FV(M_1\ M_2) &= FV(M_1) \cup FV(M_2) \\
[x := e']x &= e' \\
[x := e']y &= y \quad (x \neq y) \\
[x := e']\lambda y.e &= \lambda y.[x := e']e \quad (x \neq y \text{ かつ } y \notin FV(e)) \\
[x := e'](e_1\ e_2) &= ([x := e']e_1)\ ([x := e']e_2)
\end{aligned}
$$

図 2.3　単純型付きラムダ計算における自由変数および代入の定義

は CPS 変換する前のラムダ項には現れないとする。

自由変数に対してどのような型が割り振られているかを表す型環境は変数の有限集合から型集合への関数である Γ によって表される。

型環境 Γ においてラムダ式 e が型 T を持つという性質を以下の形の型判定として表す。

$$\Gamma \vdash e : T$$

単純型付きラムダ計算の型システムとは、型判定が成立するかどうかを判定するシステムである。次に示す型付け規則を有限回用いて型判定が導き出せるとき、その型判定は成立する。

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ var \qquad \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash (\lambda x.e) : T_1 \to T_2}\ abs \qquad \frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash (e_1\ e_2) : T_2}\ app$$

ラムダ項 e 中の自由変数の集合 $FV(e)$、e の自由変数 x を e' によって置き換える代入 $[x := e']e$ の定義は図 2.3 のように与えられる。

ここで代入を定義するにあたって必要となる約束を導入する。それは次のようなものである。

束縛変数の名前のみが異なる項は、任意の文脈で置き換え可能である。

この約束は非形式的なものであり、代入を定義するにあたって便宜的に導入した仮定である。

$$
\begin{aligned}
e := \ & n \\
& \mid (\lambda.e) \\
& \mid (e_1\ e_2)
\end{aligned}
$$

図 2.4　De Bruijn インデックスによる単純型付きラムダ計算の項の定義

このような約束は非形式的な証明に多く用いられている。非形式的な証明であれば、代入において現れる付帯条件 $x \neq y$ などについて処理することが容易であるからである。しかしながら、このような表現の単純型付きラムダ計算を Coq を用いて形式化することにはある問題が存在する。それは、例えば代入を含むような等式について変形を施す必要がある場合、上記の定義の代入では単純な等式変形とはならず、付帯条件に関しても形式的に扱う必要が出てくることにある。付帯条件に関して形式的に扱う必要があることから、自動的に行われるような、単なる計算として代入を扱うことはできないことになる。このように、このような表現による定義は形式化するにあたっては不満が残るものである。

2.2　De Bruijn インデックスによる名無し表現の単純型付きラムダ計算

2.1 節において導入した単純型付きラムダ計算の代入の定義は「束縛変数の名前のみが異なる項は、任意の文脈で置き換え可能である」という約束にもとづいていた。このような代入の定義、ひいては単純型付きラムダ計算の定義は計算機の上において形式化する上で表現のし易いものとは言いがたい。そこで今回 Coq の上で単純型付きラムダ計算を形式化するにあたって、変数の束縛を de Bruijn インデックスを使い表す。

De Bruijn インデックスを用いた表示では、変数の名前の代わりに自然数を変数を表すために使う。ラムダ抽象において、項の中のそれぞれの変数は、x や y といった名前を与えて束縛するのではなく、対応する束縛を示す数によって表される。

例えば、$\lambda x.x(\lambda y.y\ x)$ というラムダ項は de Bruijn インデックスによる表示では、$\lambda.0\ (\lambda.0\ 1)$ となる。0 は、その変数から見て最も内側の（最も近い）ラムダ抽象の束縛であることを表し、1 は 2 番目の、2 は三番目の、のように数は対応する束縛に応じて変化する。

また、対応する束縛のない変数は自由変数である。例えば $\lambda.\lambda.(0\ 2)$ のようなラムダ項について変数 0 は内側のラムダ抽象によって束縛されているが、変数 2 はどのラムダ抽象によっても束縛されておらず、自由変数である。

De Bruijn インデックスを用いるとラムダ項の文法は図 2.4 のように表される。ここで、n は自然数である。

De Bruijn インデックスにより表されるラムダ項に関する代入を定義するために、補助的なシフト $\uparrow_c^d (\cdot)$ という関数を定義する。この操作は項に現れる自由変数のインデックスを付け替える。

$$
\begin{aligned}
\uparrow_c^d (k) &= \begin{cases} k & k < c \text{ の場合} \\ k + d & k \geq c \text{ の場合} \end{cases} \\
\uparrow_c^d (\lambda.e) &= \lambda.(\uparrow_{c+1}^d e) \\
\uparrow_c^d (e_1\ e_2) &= (\uparrow_c^d e_1\ \ \uparrow_c^d e_2)) \\
[j := s]x &= \begin{cases} s & k = j \text{ の場合} \\ k & \text{それ以外の場合} \end{cases} \\
[j := s](\lambda.e) &= \lambda.[j + 1 := \uparrow_0^1 (s)]e_1 \\
[j := s](e_1\ e_2) &= ([j := s]e_1\ [j := s]e_2)
\end{aligned}
$$

図 2.5　De Bruijn インデックスによる単純型付きラムダ計算のシフトと代入の定義

このシフトを用いることで、de Bruijn インデックスによる代入 $[j := s]\cdot$ を定義することができる。このように代入を定義することで、付帯条件を気にすることなく代入を計算によって行うことができるようになる。

De Bruijn インデックスを導入したことによる型集合の定義の変更はないが、型環境は変数ではなく自然数を受け取るように変化する。

$$
\begin{aligned}
\Gamma &:= \phi \\
&\mid \Gamma, T
\end{aligned}
$$

そして、型判定に関する規則も変化する。

$$
\frac{\Gamma(n) = T}{\Gamma \vdash n : T}\ var \quad \frac{\Gamma, T_1 \vdash e : T_2}{\Gamma \vdash (\lambda.e) : T_1 \to T_2}\ abs \quad \frac{\Gamma \vdash e_1 : T_1 \to T_2 \ \ \Gamma \vdash e_2 : T_1}{\Gamma \vdash (e_1\ e_2) : T_2}\ app
$$

2.3　名前付き表現の CPS 変換と型保存定理

ラムダ項に対する、Plotkin による値呼び戦略の CPS 変換（以下 CPS 変換）[1] は関数 $[\![\cdot]\!]$ によって、値に対する関数は $\underline{\cdot}^*$ によって定義される。図 2.6 に定義を示す。

型と型環境に対する CPS 変換は $[\![\cdot]\!]$ と $\underline{\cdot}^*$ によって定義される。図 2.7 に定義を示す。

以上の定義のもとで、次のような定理がなりたつことが知られている。

定理 2.3.1（型保存定理）　型環境 Γ、ラムダ項 ϵ、型 T に対して次が成立する。

$$
\Gamma \vdash e : T \text{ ならば } \Gamma^* \vdash [\![e]\!] : [\![T]\!]
$$

Proof. $\Gamma \vdash e : T$ の導出に関する帰納法による。詳細は [14] による。　□

ここで、完全性の言明を表すのに必要な、項の間で成り立つ同値性に関して説明する。項同士の同値性を表す公理は次のように表される。ただし、M, N はラムダ項、V は値である。

$$
\begin{aligned}
[\![x]\!] &= \lambda k.k\ x \\
[\![\lambda x.e]\!] &= \lambda k.k\ (\lambda x.[\![e]\!]) \\
[\![(e_1\ e_2)]\!] &= \lambda k.[\![e_1]\!]\ (\lambda m.[\![e_2]\!]\ (\lambda n.m\ n\ k)) \\
\underline{x}^* &= x \\
\underline{(\lambda x.e)}^* &= \lambda x.[\![e]\!] \\
&\text{ただし、上記の } k, m, n \text{ はすべて fresh}
\end{aligned}
$$

図 2.6　ラムダ項と値の CPS 変換の定義

$$
\begin{aligned}
[\![T]\!] &= (\underline{T}^* \to Unit) \to Unit \\
\underline{Con}^* &= Con \\
\underline{(T_1 \to T_2)}^* &= \underline{T_1}^* \to [\![T_2]\!] \\
[\![x_1 : T_1, \cdots, x_n : T_n]\!] &= x_1 : \underline{T_1}^*, \cdots, x_n : \underline{T_n}^*
\end{aligned}
$$

図 2.7　型の CPS 変換の定義

$$
\begin{aligned}
\beta_v &: (\lambda x.M)\ V = [x := V]M \\
\eta_v &: \lambda x.V\ x = V \quad x \notin FV(V) \\
\beta_\Omega &: (\lambda x.x)\ M = M \\
&\quad (\lambda x.x\ N)\ M = M\ N \quad x \notin FV(N) \\
&\quad (\lambda x.V\ N)\ M = V\ ((\lambda x.N)\ M) \quad x \notin FV(V)
\end{aligned}
$$

図 2.8　同値性

ある公理の集合 X を認めた上で、項同士が同値であるという言明を表すのに、次のような記法を用いる。今後、これを（ラムダ項の）同値性判定と呼ぶ。

$$\lambda X \vdash M = N$$

例えば、次の同値性判定が成り立つ。

$$\lambda\beta_v \vdash (\lambda x.x)y = y$$

この同値性判定の記法を用いて、完全性定理は次のように表される。

定理 2.3.2（完全性定理）　2 つのラムダ項 e_1、e_2 について、

$$\lambda\beta\eta \vdash [\![e_1]\!] = [\![e_2]\!] \text{ ならば } \lambda\beta_v\eta_v\beta_\Omega \vdash e_1 = e_2$$

完全性定理に関する証明は後述する。

2.4 Coq を用いた名無し表現の単純型付きラムダ計算の形式化

本節では実際の Coq のプログラムとともに、de Bruijn インデックスを用いた名無し表現の単純型付きラムダ計算をどのように形式化するかを説明する。

名無し表現において、変数はそれが束縛されているラムダ抽象を指し示す数によって表されている。ゆえに、名無し表現におけるラムダ項は自然数を要素としてもつ。ラムダ項は、変数のみではなくラムダ抽象や関数適用の形も持っている。それを表すためにはラムダ項を表すような型を定義する。

ラムダ項と型の Coq での定義を図 2.9 に示す。OCaml の `type` キーワードのように、型を定義するために使える Coq のコマンドとして `Definition` と `Inductive` がある。これらを用いて、ラムダ項を表す型 `term` と型を表す型 `type` を作る。抽象化のためにラムダ項の変数のコンストラクタの引数は Coq の `nat`（自然数）型ではなく、`nat` をラップした `var` 型で表している。型変数に関しても `tvar` 型を作り、同じことをしている。

`term` のコンストラクタはそれぞれ、`EVar` が変数に、`EAbs` がラムダ抽象に、`EApp` が関数適用に対応している。なお、値を表すような型は作成しなかったが、これは `isval` 述語によって `term` 型の項が値かどうかを表すようにした。

帰納的な型を定義した場合、Coq はそれに対する帰納法の原理を同時に、自動的に定義する。例として、type を定義した時に生成される命題に関する帰納法の原理の型を以下に示す。[*2]

```
forall P : type → Prop,
(forall t : tvar, P (TVar t)) →
P TCon →
(forall t : type, P t → forall t0 : type, P t0 → P (TFun t t0)) →
P TUnit → forall t : type, P t
```

ただし、自動的に定義される帰納法の原理は、例えば相互帰納的な型であればそれに則したものが作成されるわけではない。すなわち、相互帰納的な型に適用しても、それだけでは相互帰納的な証明を行うことができないような帰納法の原理が作成されてしまう。だが、相互帰納的な型に関しては、後述する Scheme コマンドを使うことで相互帰納的な帰納法の原理を作ることができる。

型環境は型のリスト、型判定は帰納的述語として表す。`Reserved Notation` というコマンドと `Inductive` の最後に付いている `where` 節により、型環境 `G` でラムダ項 `e` が `t` を持つという型判定は `G |= e ; t` と書けるようになる（図 2.10）。

シフトと代入の定義は `Fixpoint` コマンドを使う。これは、再帰的な関数の定義のためのコマンドである。`ltb x y` は $x < y$ の結果を `bool` 型で返す関数である。`substitute` に現れる `S` は `nat` 型のコンストラクタで、後続数を表している。`S j` は `j` の後続数であることを表している（図 2.11）。

[*2] 帰納法の原理はまず型に関するものが定義され、それを用いて命題や集合に関するものが同時に定義される。

```
Definition var := nat.
Definition tvar := nat.

Inductive term :=
| EVar : var → term
| EAbs : term → term
| EApp : term → term → term.

Inductive type :=
| TVar : tvar → type
| TCon : type
| TFun : type → type → type
| TUnit : type.

Definition beq_var := beq_nat.

Definition isval (e : term) :=
  match e with
    | EVar _ ⇒ True
    | EAbs _ ⇒ True
    | EApp _ _⇒ False
  end.
```

図 2.9　ラムダ項と型の Coq での定義

2.5　コンパイラの中間表現としての CPS 変換

CPS 変換により生成される中間表現、継続渡し形式はいくつかの点で他の手法と比べて優れている。

例として、継続を陽に持たないラムダ計算を中間表現として用いることとの比較を見る。正格な値呼びプログラム言語（ML や Lisp など）では、関数の引数は関数の本体の前に評価される。しかし、継続を陽に持たないラムダ計算では必ずしもこれが成り立つ必要があるわけではない。これを行うための方法として、通常は、実引数のコピーを仮引数の出現位置それぞれに埋め込む。このことは、

- 正格に評価されれば、無限ループに陥るようなプログラムが止まってしまうかもしれない。
- 仮引数が元のプログラムで何度も使われていれば、本来は一回しか評価されない実引数が何度も評価されてしまう。
- 副作用のある言語では、実引数の副作用が関数本体の副作用のうちのいつかであったり、

```
Definition assump := list type.

Reserved Notation "G |= e ; t" (at level 80).

Inductive hasType : assump → term → type → Prop :=
(* Typing relation for simply typed lambda calclus (de buijn index) *)
| Var_ht : forall (G:assump) (x : var) (t: type),
             List.nth_error G x = value t → G |= EVar x ; t
| Lam_ht : forall (G : assump) (e : term) (t1 : type) (t2 : type),
            (t1 :: G |= e ; t2) → G |= EAbs e ; TFun t1 t2
| App_ht : forall (G : assump) (e1 e2 : term) (t1 t2 : type),
            G |= e1 ; (TFun t1 t2) → G |= e2 ; t1 → G |= (EApp e1 e2) ; t2
where "G |= e ; t" := (hasType G e t).
```

図 2.10　型判定規則

```
Fixpoint shift (d c : nat) (e : term) : term :=
  match e with
    | EVar x ⇒ if ltb x c then EVar x else EVar (d + x)
    | EAbs e ⇒ EAbs (shift d (S c) e)
    | EApp e1 e2 ⇒ EApp (shift d c e1) (shift d c e2)
  end.

Fixpoint substitute (j : nat) (s: term) (e : term) : term :=
  match e with
    | EVar x ⇒ if beq_var x j then s else EVar x
    | EAbs e ⇒ EAbs (substitute (S j) (shift 1 0 s) e)
    | EApp e1 e2 ⇒ EApp (substitute j s e1) (substitute j s e2)
  end.
```

図 2.11　シフトと代入の Coq プログラム

まったく起きなかったり、一回以上起きるかもしれない。

CPS による表現では以上の問題が発生しない。すべての関数の実引数は変数か定数かラムダ抽象であり、非自明な式（関数適用など）であったりしない。したがって、仮引数への代入は問題を起こさない。

この他にも CPS 変換、そして継続渡し形式は様々な利点を持つ。詳しい解説は、[15] に見られる。

第 3 章

名無し表現の CPS 変換と型保存定理

本章は形式化された CPS 変換の説明を行い、その後、それを用いた形式的証明の例として、CPS 変換についての型保存定理の形式化を与える。3.1 節で名無し表現の CPS 変換の定義を説明し、そのまま Coq での形式化を与える。3.2 節では型保存定理に関する Coq での形式化の方法を処理系からの出力も合わせて解説する。

3.1 名無し表現の CPS 変換とその形式化

De Bruijn インデックスを使うことで、名無し表現にしたラムダ項と値に対する CPS 変換は定義が変わり、図 3.1 にあるような形になる。項に対してどのように再帰するかというおおまかな構造は変わらないが、CPS 変換で追加されるラムダ抽象により変数が増えることからその分のシフトをしなければいけなくなる。

De Bruijn インデックスにより表されたラムダ項と値に関する CPS 変換の定義を図 3.2 に、型に関する CPS 変換の定義を図 3.3 に示す。ラムダ項に対する CPS 変換 $[\![\cdot]\!]$ は `cpstrans`、値に対する CPS 変換 $\underline{\cdot}^*$ は `cpstrans_value` に対応している。`cpstrans_value` の定義で、本来ならば存在しない `e` が `EApp e1 e2` の形をしているとき、すなわち値が関数適用の形をしているときに対するガードがある。これは、`cpstrans_value` を適用する対象は `isval` によって値であること

$$
\begin{aligned}
[\![n]\!] &= \lambda.0\ \uparrow_0^1 n \\
[\![\lambda.e]\!] &= \lambda.0\ \uparrow_0^1 (\lambda.[\![e]\!]) \\
[\![(e_1\ e_2)]\!] &= \lambda.\uparrow_0^1 [\![e_1]\!]\ (\lambda.\uparrow_0^2 [\![e_2]\!]\ (\lambda.0\ 1\ 2)) \\
\underline{n}^* &= n \\
\underline{(\lambda.e)}^* &= \lambda.[\![e]\!] \\
[\![T_1, \cdots, T_n]\!] &= \underline{T_1}^*, \cdots, \underline{T_n}^*
\end{aligned}
$$

図 3.1 De Bruijn インデックスによる CPS 変換の定義

```
Fixpoint cpstrans_value (e : term) : term :=
 match e with
   | EVar s ⇒ e
   | EAbs e' ⇒ EAbs (cpstrans e')
   | EApp e1 e2 ⇒ EVar 0     (* Useless but necessary guard for exhaustive pattern. *)
 end
with
 cpstrans (e : term) : term :=
 match e with
   | EVar s ⇒ EAbs (EApp (EVar 0) (shift 1 0 e))
   | EAbs e ⇒ EAbs (EApp (EVar 0) (shift 1 0 (EAbs (cpstrans e))))
   | EApp e1 e2 ⇒ EAbs (EApp (shift 1 0 (cpstrans e1))
                             (EAbs (EApp (shift 2 0 (cpstrans e2))
                                         (EAbs (EApp (EApp (EVar 1) (EVar 0)) (EVar 2))))))
 end.
```

図 3.2　ラムダ項と値に関する CPS 変換の Coq プログラム

確かめたものだけにするという約束を置くことで無視できる。

Unit 型は CPS 変換する前の型には現れないという前提のもとで、型と型環境に関する CPS 変換は次のように表される。型に関する CPS 変換で、$[\![\cdot]\!]$ は tycpstran_term に、$\dot{-}^*$ は tycpstrans_value に対応している。型環境に対する CPS 変換は cpstrans_context である。

3.2　Coq を用いた型保存定理の形式化

Coq を用いて表された名無し表現の単純型付きラムダ計算について、型保存定理の言明は次のように表される。

```
Lemma preservation : forall (G : assump) (e : term) (t : type) ,
                       G |= e ; t → cpstrans_context G |= cpstrans e ; tycpstrans_term t.
```

証明は、G |= e ; t の導出に関する帰納法による。導出の最後に使われた規則によって場合分けをする。

Var_ht の場合:　この場合、帰納法の仮定より、型環境 G の x 番目に型 t が存在するということを表す nth_error G x = value t が前提に追加される。

cpstrans e は簡約すると EAbs (EApp (EVar 0) (EVar (S x))) に、tycpstrans t は少し変形して TFun (TFun (tycpstrans_value t) TUnit) TUnit になる。

まず、EVar 0 が持つべき型が何であるかは容易にわかり、TFun (tycpstrans_value t) TUnit である。これは Lam_ht 規則を適用するだけでよい。これにより、型環境は

```
Fixpoint tycpstrans_term (ty : type): type :=
 let tycpstrans_value := fix tycpstrans_value (ty : type) :=
 match ty with
   | TCon ⇒ ty
   | TUnit ⇒ ty
   | TVar n ⇒ ty
   | TFun t1 t2 ⇒
     TFun (tycpstrans_value t1) (tycpstrans_term t2)
 end
 in (TFun (TFun (tycpstrans_value ty) TUnit) TUnit).

Fixpoint tycpstrans_value (ty : type) : type :=
 match ty with
   | TCon ⇒ ty
   | TUnit ⇒ ty
   | TVar n ⇒ ty
   | TFun t1 t2 ⇒
     TFun (tycpstrans_value t1) (tycpstrans_term t2)
 end.

Definition cpstrans_context (ctxt : list type) : list type :=
 List.map tycpstrans_value ctxt.
```

図 3.3　型に関する CPS 変換の Coq プログラム

TFun (tycpstrans_value t) TUnit が追加され、Fun (tycpstrans_value t) TUnit :: cpstrans_context G になる。

次に、(EApp (EVar 0) (EVar (S x))) が持つべき型がなんであるかを考えなければならないが、項の形から、それは関数型に別の型を適用したものでなければならないとわかる。App_ht 規則を用いるが、eapply タクティクで、具体的な関数型の引数の型はわからないままにする[*1]。そして、証明しなければいけないことは 2 つ (EVar 0) と (EVar (S x)) の型判定に分解する。

前者に関しては容易である。型環境の先頭に EVar 0 に対応する型 TFun (tycpstrans_value t) TUnit が存在するため、Var_ht 規則を適用すれば即座に証明可能である。なおここで証明が終わると同時に、関数型の引数の型も判明する。

後者に関してはラムダ抽象により型環境に追加された項を考慮にいれる必要がある。だが、型判定 G |= e ; t の内部で、e と G の間に構文的なつながりはなく、述語 hasType に

[*1] ただし、具体的な項を与えてやらなければ全体の証明（今回は型保存定理）を終えることはできない。

よってのみその２つはつながっている。

ここで、別個に証明した補題を用いる。それが shift_context と名付けられた補題である。この補題は、ラムダ抽象により一つの変数が導入された型判定は、ある条件のもとそれを無視した型判定が成り立っていれば成り立つというものである。

```
Lemma shift_context : forall (e :term) (ty1 ty2 : type) (tyls1 tyls2 : list type) ,
                tyls1 ++ tyls2 |= e; ty2 →
                tyls1 ++ (ty1 :: tyls2) |= shift 1 (length tyls1) e; ty2.
```

この補題と、CPS 変換された型環境は元の型環境の順番を保存している性質を用いることで、EVar (S x) に関する型判定を証明することができる。

Lam_ht の場合: cpstrans e は簡約すると EAbs (EApp (EVar 0) (EAbs (shift 1 1 (cpstrans e)))) に、tycpstrans_term t は TFun (TFun (TFun (tycpstrans_value t1) (tycpstran_term t2)) TUnit) TUnit に簡約される。

この場合、帰納法の仮定より次の２つの前提が追加される。型 t1 **や** t2 が帰納法によって新たに導入されていることがわかる。

```
 t1 ::  G |= e;  t2
cpstrans_context (t1 :: G) |= cpstrans e; tycpstrans_term t2
```

証明すべき型判定のラムダ項と型を見ると、Lam_ht を使えばよいことがわかり、それにより導入される型は (TFun (TFun (tycpstrans_value t1) (tycpstran_term t2)) TUnit) に決まる。

次はラムダ項の形から、App_ht を用いる必要があるが、ここでも先に証明した Var_ht の場合のように eapply タクティクを用いて、関数型の引数の型をわからないままにして、先送りする。すると証明すべき型判定は２つに分解される。その２つに対して、ラムダ項と型に応じて適宜 Var_ht か、Lam_ht と shift_context を適用することで証明ができる。

App_ht の場合　次の４つの前提が帰納法の仮定より追加される。型 t1 **や** t2、項 e1 **や** e2 が帰納法によって新たに導入されていることがわかる。

```
G |= e1; TFun t1 t2
G |= e2; t1
cpstrans_context G |= cpstrans e1; tycpstrans_term (TFun t1 t2)
cpstrans_context G |= cpstrans e2; tycpstrans_term t1
```

cpstrans e は簡約すると

```
EAbs (EApp (shift 1 0 (cpstrans e1))
           (EAbs (EApp (shift 2 0 (cpstrans e2))
                       (EAbs (EApp (EApp (EVar 1) (EVar 0)) (EVar 2))))))
```

になり、tycpstrans_term t は

```
TFun (TFun (tycpstrans_value t2) TUnit) TUnit
```

に簡約される。基本的には Lam_ht の場合のように適宜式を変形しつつラムダ項と型の形に応じて規則を適用するだけでよいが、２つのラムダ抽象があることから Var_ht

で使った shift_context に似た、だが 2 つの変数に関する補題が必要になる。それが shift2_context である。

```
Lemma shift2_context : forall (e :term) (ty1 ty2 ty3 : type) (tyls1 tyls2 : list type) ,
                tyls1 ++ tyls2 |= e; ty3 →
                tyls1 ++ ty1 :: ty2 :: tyls2 |= shift 2 (length tyls1) e; ty3.
```

shift2_context の証明に必要な補題は shift_context のものと共有されている。この補題のもとで、2 つのラムダ抽象の内側にある変数、これは $\uparrow_0^2$ が適用されている、この変数について帰納法により導入された仮定を使える形に、型判定を変形することができる。

以上で、形式化により、型保存定理の証明を Coq で完成させることができる。ラムダ項、型集合、型判定など必要な定義を含めた型保存定理の Coq の証明スクリプトは 318 行であった。

第 4 章

名無し表現による CPS 変換の完全性の形式化

コンパイラなどの言語処理系を通すことによるプログラムの変換は、変換前のプログラムが意図したように変換後のプログラムも動作することを要求する。すなわち、ある意味でプログラム変換について意味が保存されることを要求する。CPS 変換についてそれを表す性質は、例えば健全性と完全性という二つの性質が挙げられる。

ラムダ計算について定義される健全性は、公理として β_v を選んだときラムダ項の同値性判定が CPS 変換によって保存されることを表す。健全性の証明は [1] に見られ、難解なものではない。

完全性は 2 章で与えたような定義である。これを再掲すると、2 つのラムダ項 e_1、e_2 について、

$$\lambda\beta\eta \vdash [\![e_1]\!] = [\![e_2]\!] \text{ ならば } \lambda\beta_v\eta_v\beta_\Omega \vdash e_1 = e_2$$

が成り立つことを言う。この定理が成り立つことから、CPS 変換された項の同値性の公理 $\beta\eta$ に対応する元の項での同値性の公理 $\beta_v\eta_v\beta_\Omega$ を用いて、CPS 変換することなしに、CPS 変換された項での $\beta\eta$ 簡約に対応するような最適化を行うことができるようになる。

完全性の証明は [16, 17] に見られるが、健全性と比べて難解である。完全性の証明の困難な点は、これが CPS 文法と逆 CPS 変換と呼ばれるものを定義し、それらを用いて議論を行うことにある。CPS 文法は、CPS 変換されたラムダ項に関してそれらすべてを包含するような集合として定義し、CPS 文法の式から元の項に戻す関数として逆 CPS 変換は定義される。証明は、この CPS 文法の中で、継続を表す変数だけが持つ線形性という性質を用いている。このことから、完全性を形式化する際に変数、特に継続の取り扱いについて注意を払わなければならないということがわかる。

4.1 節で完全性の証明に必要な二つの補題を示す。4.2 節では、その二つの補題のうち、一方の証明について必要であり、そして鍵となる定理の非形式的な証明の概要を名前付き表現で与える。そして 4.3 節でその証明の Coq を用いた形式化について説明をする。4.4 節では Coq を用いた形式化の際に現れた問題について解説する。

$$
\begin{array}{ll}
T := \lambda k.P & \mid (V\ V) \\
P := K\ V & \mid (T\ K) \\
K := k & \mid \lambda x.P \\
V := x & \mid \lambda x.T
\end{array}
$$

図 4.1　CPS 文法の定義

4.1　完全性の証明のために

完全性を証明するためには、大きく分けて二つの補題が必要になる。

次のように表される補題であり、本章で詳しく扱う完全性の証明に鍵となる補題を用いてこの補題は証明される。

補題 4.1.1　任意の CPS 項 M と N について、

$$\lambda\beta\eta \vdash M = N \text{ ならば } \lambda\beta_v\eta_v\beta_\Omega \vdash M^{-1} = N^{-1}$$

また、これ以外にももう一つ補題が必要となる。それは逆 CPS 変換が確かに CPS 変換の逆であることの証明である。すなわち、

補題 4.1.2（逆 CPS 変換と CPS 変換）　任意のラムダ項 e について、

$$\lambda\beta_v\eta_v\beta_\Omega \vdash [\![e]\!]^{-1} = e$$

が成り立つことである。

この二つの補題により完全性の証明は完成する。前者を導き出すために使う補題を本章では重点的に解説するが、それはこの補題が他の補題と比べ非形式的な証明が複雑で、形式化にも困難が予想されるからである。

4.2　名前付き表現による完全性の証明

CPS 変換により変換されたラムダ項は、もちろんラムダ項の文法によって表されるが、CPS 文法という文法によって表すこともできる。今までに定義したラムダ項と CPS 変換についての CPS 文法は図 4.1 にある。このような文法で表される項を今後 CPS 項と呼ぶ。ここで以前と同じように x は加算無限個の変数の集合 Var を代表するメタ変数である。k は継続を表す変数である。

このように定義された文法は、それだけで次のような性質を CPS 項に与える。

CPS 項は β 簡約について閉じている。すなわち、β 簡約基のある CPS 項を β 簡約した結果も CPS 文法によって表される式である、すなわち CPS 項である。

$$
\begin{array}{lrcl}
T^{-1}: & (\lambda k.P)^{-1} & = & P^{-1} \\
 & (V_1\ V_2)^{-1} & = & V_1^{-1}\ V_2^{-1} \\
P^{-1}: & (K\ V)^{-1} & = & K^{-1}\ V^{-1} \\
 & (T\ K)^{-1} & = & K^{-1}\ T^{-1} \\
K^{-1}: & k^{-1} & = & \lambda x.x \\
 & (\lambda x.P)^{-1} & = & \lambda x.P^{-1} \\
V^{-1}: & x^{-1} & = & x \\
 & (\lambda x.T)^{-1} & = & \lambda x.T^{-1}
\end{array}
$$

図 4.2　逆 CPS 変換の定義

命題 4.2.1　継続 k について次のような性質が成り立つ。

- k は T、V に自由に出現することはない。
- P の中で、継続に関するラムダ抽象 λk にただ一つだけ対応する k が出現する。このような性質を、k が P の中で線形であるという。

2 番目の性質を例により説明する。次のような CPS 項を考える。見やすさのため、適宜、変数には $'$ を付与している。

$$\lambda k.(k\ (\lambda x.(\lambda k'.k'\ x)))$$

この CPS 項の一番外側の継続 λk に関するラムダ抽象に対して、すぐにその継続が出現している。CPS 項を走査していくと、次の継続に関するラムダ抽象 $\lambda k'$ に対してもその内側の項で継続が出現していることがわかる。

K の $\lambda x.P$ で束縛されている変数 x は CPS 変換によって導入された、元の項にはなかったラムダ抽象による変数を表す。

CPS 項に対して、CPS 変換した項を元に戻す変換、逆 CPS 変換を定義することができる（図 4.2)。これは T、P、K、V の各変数ごとに定義され、相互再帰的な定義になっている。

4.1 節で示した一つ目の補題の証明について重要な補題として次の 2 つがあげられる。

補題 4.2.2（逆 CPS 変換と値の代入に関する性質）　値 v を CPS 項に代入した結果を逆 CPS 変換したものと、CPS 項を逆 CPS 変換した結果のラムダ項に、値 v を逆 CPS 項変換したものを代入した結果について次のような同値性判定が成り立つ。

$$
\begin{aligned}
\lambda\beta_v\eta_v\beta_\Omega &\vdash ([x := v]T)^{-1} = [x := v^{-1}]T^{-1} \\
\lambda\beta_v\eta_v\beta_\Omega &\vdash ([x := v]P)^{-1} = [x := v^{-1}]P^{-1} \\
\lambda\beta_v\eta_v\beta_\Omega &\vdash ([x := v]K)^{-1} = [x := v^{-1}]K^{-1} \\
\lambda\beta_v\eta_v\beta_\Omega &\vdash ([x := v]V)^{-1} = [x := v^{-1}]V^{-1}
\end{aligned}
$$

補題 4.2.3（逆 CPS 変換と継続の代入に関する性質）　継続 k を CPS 項に代入した結果を逆 CPS 変換したものと、その継続 k を逆 CPS 変換したものを逆 CPS 変換したラムダ項に代入した結果

について、次のような式が成り立つ。。

$$\lambda\beta_v\eta_v\beta_\Omega \vdash ([k := K]P)^{-1} = K^{-1}\ P^{-1}$$
$$\lambda\beta_v\eta_v\beta_\Omega \vdash ([k := K]K_0)^{-1} = \lambda x.K^{-1}\ (K_0^{-1}\ x)$$

このうち、特に補題 4.2.3 は、単純に値の代入と逆 CPS 変換が可換であることを示す補題 4.2.2 とくらべて非自明な形である。以後、この証明について詳しく解説を行う。

Proof. この補題の証明は P と K_0 に関する同時帰納法によって行われる。なお、簡単のために同値性判定の等式部分だけを記述する。

$P = (K_1\ V)$ **のとき**: 帰納法の仮定より、任意の K について次の式が成り立つ。

$$(K_1[k := K])^{-1} = \lambda x.K^{-1}(K_0^{-1}\ x)$$

このとき、

$$\begin{aligned}(P[k := K])^{-1} &= ((K_1\ V)[k := K])^{-1}\\ &= ((K_1[k := K])\ V)^{-1}\\ &= (K_1[k := K])^{-1}\ V^{-1}\\ &= (\lambda x.K^{-1}(K_1^{-1}\ x))V^{-1}\\ &= K^{-1}\ (K_1^{-1}\ V^{-1})\end{aligned}$$

であり、かつ

$$\begin{aligned}K^{-1}\ P^{-1} &= K^{-1}\ (K_1\ V)^{-1}\\ &= K^{-1}\ (K_1^{-1}\ V^{-1})\end{aligned}$$

なので、$(P[k := K])^{-1} = K^{-1}\ P^{-1}$ が成立する。

$P = (T\ K_1)$ **のとき**: 帰納法の仮定より、任意の K について次の式が成り立つ。

$$(K_1[k := K])^{-1} = \lambda x.K^{-1}(K_0^{-1}\ x)$$

このとき、

$$\begin{aligned}(P[k := K])^{-1} &= ((T\ K_1)[k := K])^{-1}\\ &= (T\ (K_1[k := K]))^{-1}\\ &= (K_1[k := K])^{-1}\ T^{-1}\\ &= (\lambda x.K^{-1}(K_1^{-1}\ x))T^{-1}\\ &= K^{-1}\ (K_1^{-1}\ T^{-1})\end{aligned}$$

であり、かつ

$$\begin{aligned}K^{-1}\ P^{-1} &= K^{-1}\ (T\ K_1)^{-1}\\ &= K^{-1}\ (K_1^{-1}\ T^{-1})\end{aligned}$$

なので、$(P[k := K])^{-1} = K^{-1}\ P^{-1}$ が成立する。

$K_0 = k$ **のとき**:　帰納法の仮定より、任意の K について次の式が成り立つ。

$$(P[k := K])^{-1} = K^{-1}\ P^{-1}$$

このとき、

$$(K_0[k := K])^{-1} = (k[k := K])^{-1} = K^{-1}$$

であり、かつ

$$\begin{aligned}(\lambda x.K^{-1}\ (K_0^{-1}\ x))^{-1} &= \lambda x.K^{-1}\ (k^{-1}\ x)\\ &= \lambda x.K^{-1}\ ((\lambda x.x)x)\\ &= \lambda x.K^{-1}\ x\\ &= K^{-1}\end{aligned}$$

なので、$(K_0[k := K])^{-1} = (\lambda x.K^{-1}\ (K_0^{-1}\ x))^{-1}$ が成立する。

$K_0 = \lambda x.P$ **のとき**:　帰納法の仮定より、任意の K について次の式が成り立つ。

$$(P[k := K])^{-1} = K^{-1}\ P^{-1}$$

このとき、

$$\begin{aligned}(K_0[k := K])^{-1} &= ((\lambda x.P)[k := K])^{-1}\\ &= (\lambda x.P[k := K])^{-1}\\ &= \lambda x.(P[k := K])^{-1}\\ &= \lambda x.K^{-1}P^{-1}\end{aligned}$$

であり、かつ

$$\begin{aligned}(\lambda x.K^{-1}\ (K_0^{-1}\ x))^{-1} &= \lambda x.K^{-1}\ ((\lambda x.P)^{-1}\ x)\\ &= \lambda x.K^{-1}\ ((\lambda x.P^{-1})x)\\ &= \lambda x.K^{-1}\ P^{-1}\end{aligned}$$

なので、$(K_0[k := K])^{-1} = (\lambda x.K^{-1}\ (K_0^{-1}\ x))^{-1}$ が成立する。

以上のことから、補題 4.2.3 が成り立つ。　□

4.3　Coq による補題の形式化

Coq を用いて補題を形式化するにあたって、CPS 項、逆 CPS 変換、継続の代入をそれぞれ de Bruijn インデックスを用いて表した。特に CPS 項の定義を図 4.3 に示す。

De Bruijn インデックスによる CPS 項の定義では、`tterm` は T に、`pterm` は P に、`continuation` は K に、`val` は V に対応している。それぞれの型のコンストラクタは、先に示した CPS 文法の形に沿っている。このように定義した CPS 項では、継続がもはや変数ではなく

```
Inductive
tterm :=                          (* 'T' *)
| TAbs : pterm → tterm
| TApp : val → val → tterm
with
pterm :=                          (* 'P' *)
| PApp1 : continuation → val → pterm
| PApp2 : tterm → continuation → pterm
with
continuation :=                   (* 'K' *)
| KVar : continuation
| KAbs : pterm → continuation
with
val :=                          (* 'V' *)
| VVar : var → val
| VAbs : tterm → val.
```

図 4.3　CPS 項の Coq での定義

なっている。これは、継続が満たす性質 4.2.1 より正当化される。

この補題の証明は非形式的な証明と同じように同時帰納法によって行われる。Coq には標準で同時帰納法を扱うための機能として、Scheme コマンドが存在する。Scheme コマンドを用いることで、相互帰納的な型に関して相互帰納的な帰納法の原理を作ることができる。この補題の証明で用いる相互帰納的な帰納法の原理は、次のようなコマンドで生成される。

```
Scheme pterm_cps_ind := Induction for pterm Sort Prop
with continuation_cps_ind := Induction for continuation Sort Prop.
```

生成される帰納的原理の型は continuation_cps_ind は

```
forall (P : pterm → Prop) (P0 : continuation → Prop),
(forall c : continuation, P0 c → forall v : val, P (PApp1 c v)) →
(forall (t : tterm) (c : continuation), P0 c → P (PApp2 t c)) →
P0 KVar →
(forall p : pterm, P p → P0 (KAbs p)) →
forall c : continuation, P0 c
```

という形をしている。pterm_cps_ind は

```
forall (P : pterm → Prop) (P0 : continuation → Prop),
(forall c : continuation, P0 c → forall v : val, P (PApp1 c v)) →
(forall (t : tterm) (c : continuation), P0 c → P (PApp2 t c)) →
P0 KVar →
```

```
(forall p : pterm, P p → P0 (KAbs p)) →
forall p : pterm, P p
```

となり、最後の結論の部分が異なる以外は非常に似通った形をしている。

4.4 逆 CPS 変換の形式化に存在する問題

本論文の単純型付きラムダ計算の形式化では、変数の表示に de Bruijn インデックスを用いている。したがって、T についての次の逆 CPS 変換

$$(\lambda k.P)^{-1} = P^{-1}$$

は、そのラムダ抽象がなくなった影響を P の内部の変数に伝搬させる必要がある。例えば T の中で出現し、しかも自由である変数に関しては、そのインデックスを一つ減らす必要がある。

上のような形のラムダ抽象（今後、継続のラムダ抽象と呼ぶ）がなくなった影響を伝搬させるには、逆 CPS 変換が相互再帰的な定義であることに注意が必要である。素朴に考えると、次のような手法が考えられる。まず逆 CPS 変換を二つの関数に分割する。

1. はじめの関数は、継続のラムダ抽象がなくなる影響を伝搬させる。このときそれ以外のことはしない、すなわち変化するのは継続のラムダ抽象の内部にある継続でない変数のみである。この関数は二つの引数をもつ。ひとつは CPS 項であり、もう一つは自然数のリストである。自然数のリストの中の一つ一つの数は、それまでに見つかった継続のラムダ抽象がなくなる影響を適用するかどうかの基準になっている。この関数は引数の CPS 項が継続のラムダ抽象の形であったとき、その内部の P について、今そのときに見ている継続のラムダ抽象をなくした影響を伝搬させるように指令を追加する。例えば次のような CPS 項の逆 CPS 変換を考える。

   $$(\lambda k.(k\ (\lambda x.(\lambda k'.k'\ x))))^{-1} = (\lambda y.y)\ (\lambda x.(\lambda y.y)\ x)$$

   これは de Bruijn インデックスにより表すと次のようにならなければならない。

   $$(\lambda.(0\ (\lambda.(\lambda.0\ 1)))) = (\lambda.0)\ (\lambda.(\lambda.0)\ 0)$$

   継続でない変数は変数 x である。この変数は de Bruijn インデックスで表したときは、外側の継続のラムダ抽象がなくなる影響を受けないが、内側のものからは影響を受ける。したがって、de Bruijn インデックスにおいて逆 CPS 変換を施した後、変数 x に対応するものは 0 になるのである。
2. 二番目の関数ははじめの関数以外の操作をすべて行う。

このようにして分割された関数によって逆 CPS 変換を定義し、形式化を行うことは原理的には不可能ではない。しかしながら、このような関数に関して推論を行い、証明を与えるには一番目の関数の性質を形式化する必要がある。関数の定義がリストと CPS 項の二つの帰納的型を扱うことから、これは非常に複雑になる。

$$T := \kappa k.P \mid (V\ V)$$
$$\cdots$$

図 4.4　改良された CPS 文法の定義（一部）

問題は継続のラムダ抽象を除去する際、その影響を伝搬させる時に起こる。これは継続のラムダ抽象とそれ以外のラムダ抽象とが、インデックスを共有していることが原因である。ゆえに、継続のラムダ抽象とそれ以外のラムダ抽象を区別すること、すなわち名前空間を分離することが解決の糸口である。例えば次のような CPS 項を考える。

$$\lambda k.(k\ (\lambda x.(\lambda k'.k'\ x)))$$

De Bruijn インデックスで表すと、

$$\lambda.(0\ (\lambda.(\lambda.0\ 1)))$$

である。この CPS 項には二つの継続のラムダ抽象と一つのそれ以外のラムダ抽象が含まれている。ここで、継続のラムダ抽象を特別に表すような記法 κ を CPS 文法に導入する（図 4.4）。これを用いて先の項を表すと、

$$\kappa k.(k\ (\lambda x.(\kappa k'.k'\ x)))$$

となる。これを de Bruijn インデックスで表すには、継続でない変数のインデックスは、継続のラムダ抽象を無視し、逆に継続の変数は継続でないラムダ抽象を無視すればよい。また、変数にはそれが継続を指すか、それ以外の変数であるかを表す情報が付属しているとする。それをそれぞれ n_κ と n_λ で表す。すると、先の CPS 項は次のように表せる。

$$\kappa.(0_\kappa\ (\lambda.(\kappa.0_\kappa\ 0_\lambda)))$$

このように定義した CPS 項では、逆 CPS 変換で継続のラムダ抽象をなくした影響を伝搬させる必要がない。

ここで定義される、継続以外の変数のシフトを $\Uparrow$（定義は図 4.5）で、継続の代入を $\{\cdot\}$（定義は図 4.6）で表す。また、非形式的に n_κ と n_λ と表していた変数の区別は、形式的には CPS 項の型の定義による。すなわち、`VVar` と `KVar` により、変数は区別される。

現在の Coq での代入やシフト、CPS 変換の定義は、非形式的な問題に対する理解によって正当化されている。したがって、より厳密性を求めるのならば、この記法それ自体が元の記法、すなわち標準的な de Bruijn インデックスによるものと表現するものが同じであるという証明を与え、そして形式化しなければならない。

それを行うためには、本節で導入したような記法と標準的な de Bruijn インデックスとが、相互に導き出せる、すなわち、全単射である写像を構成できることを示さなければならないと思われる。その証明と形式化は、結局のところ本節では避けて通った問題そのものにほかならない。

$$
\begin{array}{lrl}
T: & \Uparrow_c^d (\kappa.P) = & \kappa.\, \Uparrow_c^d P \\
 & \Uparrow_c^d (V_1\ V_2) = & \Uparrow_c^d V_1\ \ \Uparrow_c^d V_2 \\
P: & \Uparrow_c^d (K\ V) = & \Uparrow_c^d K\ \ \Uparrow_c^d V \\
 & \Uparrow_c^d (T\ K) = & \Uparrow_c^d T\ \ \Uparrow_c^d K \\
K: & \Uparrow_c^d 0_\kappa = & 0_\kappa \\
 & \Uparrow_c^d (\lambda x.P) = & \lambda x.\, \Uparrow_{c+1}^d P \\
V: & \Uparrow_c^d x = & x\,(x < c \text{ の場合}) \quad x + d\,(x \geq c \text{ の場合}) \\
 & \Uparrow_c^d (\lambda x.T) = & \lambda x.\, \Uparrow_{c+1}^d T
\end{array}
$$

図 4.5　CPS 項に関するシフトの定義

$$
\begin{array}{lrl}
P: & \{\cdot := K'\}(K\ V) = & (\{\cdot := K'\})K\ V \\
 & \{\cdot := K'\}(T\ K) = & T\ (\{\cdot := K'\}K) \\
K: & \{\cdot := K\}0_\kappa = & K \\
 & \{\cdot := K\}(\lambda x.P) = & \lambda x.\{\cdot := \Uparrow_0^1 K\}P
\end{array}
$$

図 4.6　CPS 項に関する代入の定義

この問題を除けば、補題 4.2.3 の形式化が完了できる。Coq の証明スクリプトは 663 行であった。

第 5 章

関連研究

序論でも述べたように、CPS 変換を定理証明支援系、特に Coq を用いて形式化することは、すでに様々な形でなされてきた [7, 8, 9]。

[7, 8] でも CPS 変換の形式化がなされているが、これらでも四章の最後で与えた問題が存在している。すなわち、証明を簡単にする手法として、名前空間の分離を導入したとしても、結局のところその手法自体の正当性を証明しなければならないという問題である。[9] は CPS 変換が主題ではないが、検証されたコンパイラに関するケーススタディの中で CPS 変換に関する記述がある。

Coq 以外には、[10] のように Isabelle/HOL による形式化が存在する。著者らは Plotkin による値呼び戦略に関する CPS 変換以外にも複数の CPS 変換を形式化しているが、de Bruijn インデックスとは違う手法により変数を表現している。

第 6 章

結論

6.1 まとめ

本論文では、de Bruijn インデックスを用いて表された名無し表現の単純型付きラムダ計算と、それに対する CPS 変換を Coq を用いて形式化し、CPS 変換に関する性質のうち、型保存定理と完全性の証明に鍵となる補題の形式化をした。

型保存定理の証明の形式化はその補題の証明を含めて完了している。

完全性の証明の形式化では、CPS 文法を de Bruijn インデックスを素朴に用いて表すことによって、逆 CPS 変換の中で現れる継続のラムダ抽象の除去に関して形式化が複雑になり、証明が困難になるという問題が発生することがわかった。そして、この問題を、継続について特別な取り扱いをすることで継続とそれ以外の変数の名前空間を分離するという手法により解決した。

6.2 今後の課題

まず、完全性の形式化を完了させることが課題であるが、残りの補題は、今回重点的に解説を行った補題に比べて比較的容易に証明できると思われる。

また形式化した CPS 項や CPS 項への継続の代入の定義に関して、改良すべき点が存在している。非形式的な性質 4.2.1 にもとづいて、前者では継続の変数をコンストラクタにより表しており、後者では T と V への継続の代入それ自体が定義されないとしている。この、形式化が依存している性質を形式的定義から導き出すことが改良として考えられる。

謝辞

本研究に際して、指導教員の亀山幸義教授、海野広志助教には、手厚いご指導ならびに多くの有益なご助言を賜りましたことを深く感謝申し上げます。特に、亀山教授には、丁寧かつ熱心なご指導や貴重なご助言を賜りました。心より感謝申し上げます。また、お世話になっているプログラム論理研究室の皆様にも感謝申し上げます。

参考文献

[1] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, Vol. 1, No. 2, pp. 125 – 159, 1975.

[2] Gerald Jay Sussman and Guy L Steele Jr. Scheme: An interpreter for extended lambda calculus. In *MEMO 349, MIT AI LAB*, 1975.

[3] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[4] Yves Bertot, Pierre Castran, Grard (informaticien) Huet, and Christine Paulin-Mohring. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004. Donnes complmentaires http://coq.inria.fr.

[5] Thomas C. Hales. The flyspeck project. https://code.google.com/p/flyspeck/.

[6] Georges Gonthier. Formal proof—the four-color theorem. *Notices of the American Mathematical Society*, Vol. 55, No. 11, pp. 1382–1393, 2008.

[7] Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, Vol. 4790 of *Lecture Notes in Computer Science*, pp. 211–225. Springer, 2007.

[8] Watanabe Yuki. Study on proof of type preservation in CPS transformation (CPS 変換における型保存の証明に関する研究). Master's thesis, Tokyo University, 2011.

[9] A.J. Chlipala and Berkeley University of California. *Implementing Certified Programming Language Tools in Dependent Type Theory*. University of California, Berkeley, 2007.

[10] Yasuhiko Minamide and Koji Okuma. Verifying CPS transformations in Isabelle/HOL. In *MERLIN '03: Proceedings of the 2003 workshop on Mechanized reasoning about languages with variable binding*, pp. 1–8, New York, NY, USA, 2003. ACM Press.

[11] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[12] 横内寛文. プログラム意味論 (情報数学講座). 共立出版, 6 1994.

[13] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with applications to the church-rosser theorem. *Indagationes*

Mathematicae (Koninglijke Nederlandse Akademie van Wetenschappen), Vol. 34, No. 5, pp. 381–392, 1972. http://www.win.tue.nl/automath/archive/pdf/aut029.pdfElectronic Edition.

[14] AlbertR. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi. In Rohit Parikh, editor, *Logics of Programs*, Vol. 193 of *Lecture Notes in Computer Science*, pp. 219–224. Springer Berlin Heidelberg, 1985.

[15] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[16] Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pp. 177–188, New York, NY, USA, 2003. ACM.

[17] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pp. 288–298, New York, NY, USA, 1992. ACM.