# Agda による依存型付き λ 計算の CPS 変換の実装

叢悠悠, 浅井健一

お茶の水女子大学

{so.yuyu, asai}@is.ocha.ac.jp

**概 要**　継続とは、残りの計算を表す概念である。プログラムの中で継続を扱いたいときは、プログラム全体を継続渡し形式（continuation-passing style; CPS）に変換するという方法がある。これまで、CPS 変換は主に単純型や多相型を持つ体系で考えられてきており、依存型を含む体系ではあまり議論されていない。そこで、本研究では Π 型が入った λ 計算に対する CPS 変換を定理証明支援系 Agda で実装することを試みる。Agda を用いると、正しく型が付いた項を表現することができるため、CPS 変換のプログラムは、変換の前後において型が保存されることの証明とみなすことができる。Π 型を含む λ 計算では、型と項の両方に対する代入と、項を型に持ち上げるコンストラクタが必要となる。本稿では、これらを含む体系に対する CPS 変換を定義し、それを実装した結果を示す。

## 1　はじめに

継続とは、ある時点における残りの計算のことである。たとえば、$1 + (2 * 3)$ という式の $2 * 3$ を計算しているときの継続は、「$2 * 3$ の結果を受け取ったら、それに 1 を足す」という計算である。

プログラムの中で継続を扱えるようになると、実行の流れを効率よく制御することができる [11]。その一例として、例外処理がある。例外が発生したときに、その時点の継続を破棄することで、無駄な計算をせずにプログラムを終了することができる。もう一つの例は、バックトラックである。たとえば、探索問題において、ある選択のもとで目的の解が見つからなかったときに、その選択をした時点の継続を別の選択肢で呼び出す、というプログラムを書くことができる。

プログラムの中で継続を扱うためには、継続を明示的に表す手段が必要である。その 1 つが、プログラム全体を継続渡し形式（continuation-passing style; CPS） [18] に変換する、というものである。CPS のプログラムでは、すべての計算が継続を受け取る形となり、各部分式の結果が得られたあとに、どのような計算を行うかが明示的になる。この性質を用いると、実行の流れを自由に指定することができるため、プログラムをコンパイルするときに、CPS で書かれた中間言語を採用することがある [4]。

これまで、CPS 変換は主に単純型 [17] や多相型 [15, 5] が入った体系で考えられてきた。これらの体系においては、CPS 変換の正当性や型の保存性などが示されている。CPS 変換を用いてコンパイラを実装する際に、こういった性質はコンパイラの安全性を保証するものとなる。一方、依存型を含む体系での CPS 変換に関する議論はまだ少ない。Barthe ら [7, 8] は Π 型を持つ λ 計算に対する CPS 変換を定義し、その性質に関する証明も行っているが、実装は示しておらず、実際に彼らの定義に従って Coq [9] のような依存型を持つ言語のプログラムを CPS 変換できるかどうかは明らかでない。

本研究では、Π 型を含む λ 計算に対して、型が保存される CPS 変換を Agda [19] で実装することを試みる。Agda は定理証明支援系の 1 つであり、依存型を持つプログラミング言語としてもとらえられる。依存型を持つメタ言語を用いると、正しく型が付いた項のみを表現できる対象言語を定義することが可能である [3]。型付きの項を受け取ったら、その型に適切な変換を適用した型を持つ項を返すような CPS 変換を定義できれば、変換の前後において型が保存されることが示される。

λ 計算を依存型で拡張する方法は 2 つある。1 つは、型の中に kind （型の型）を追加し、その型を持つ項を型レベルに持ち上げるコンストラクタを導入するというものである [2]。たとえば、$(\lambda x.\, x) @ y$ という項を、型に埋め込むコンストラクタに渡すことで、型レベルの関数適用とみなす、という方法である。もう 1 つは、kind を型とは別のカテゴリとして定義し、型の定義に変数や λ 抽象などのコンストラクタを追加するというものである [7]。この方法に従うと、先ほどの例は型レベルの変数 $x', y'$、λ 抽象 $\lambda'$、関数適用 $@'$ を使って $(\lambda' x'.\, x') @' y'$ と表され、これがそのまま型とみなされる。2 つの方法のうち、前者の方法で拡張した言語については、Agda による型付きの表現が与えられており、本研究ではこちらを採用する。[2] ではさらに、項と型に対する代入を明示的なコンストラクタ [1] として扱っている。代入のコンストラクタは、関数適用の型付けなどで必要となる。本稿では、明示的な代入と、項を型に埋め込むコンストラクタを含む言語に対する CPS 変換を定義し、それを実装した結果を示す。

本論文の構成は以下の通りである。2 節では、単純型付き λ 計算を用いて、型付きの項が Agda でどのように表現されるかを解説する。3 節では、単純型付き λ 計算に対する call-by-name の CPS 変換を定義する。4 節では Π 型を導入し、依存型が入ると何を考慮しなければならないのかを説明する。5 節では Π 型を含む対象言語を Agda で定義し、6 節でその言語に対する CPS 変換を与える。7 節で関連研究を紹介し、8 節でまとめと今後の課題について述べる。

なお、本研究で実装したプログラムは `https://github.com/YouyouCong/type-preserving-cps` に公開している。

## 2 Agda による単純型付き λ 計算の定義

依存型を導入する前に、Agda において単純型付き λ 計算がどのように定義されるかをみてみよう。図 1 の構文を考える。型はベースとなる int か、関数型である。コンテキストは空であるか、型が 1 つ以上入ったものである。項は変数、λ 抽象、関数適用からなる。変数の表現には de Bruijn index [13] を用いる。de Bruijn index は、その変数が内側から何番目の λ に束縛されているかを表す。たとえば、項 $\lambda x.\, \lambda y.\, x @ y$ は $\lambda.\, \lambda.\, 1 @ 0$ と表される。型付け規則において、$\Gamma \vdash t : A$ は「項 $t$ は $\Gamma$ のもとで $A$ 型を持つ」と読む。

図 1 に示した構文は、Agda において図 2 のように定義することができる。まず、型 STy を定義し、コンテキスト SCon を型の列として定義する。変数は `_∋_` によって定義される。z は de Bruijn index の 0 に対応しており、コンテキスト Γ, A の先頭要素 $A$ を指す。s は index に 1 を足す操作に対応しており、コンテキスト Γ の $i$ 番目の要素は、コンテキスト Γ, B の $i+1$ 番目の要素になることを表している。項を表す型 STm は、コンテキストと型を受け取る関数として定義される。これにより、「コンテキスト Γ のもとで A 型を持つ項」を表現することができる。たとえば、λ 抽象に対応するコンストラクタ lam は、Γ の先頭に型 A が加わったコンテキストのもとで型 B を持つ項を受け取ったら、Γ のもとで Arrow A B 型を持つ項を構成する。関数適用を表す app は、1 つ目の引数として Arrow A B 型の項を、2 つ目の引数として A 型の項を要求し、全体として B 型の項となる。なお、型 STm Γ A において、STm は「Γ と A で index されている」という。

このように定義された構文では、正しく型が付く項のみが表現される。たとえば、以下の term1 は $\bullet, x : int \vdash (\lambda y.\, y) @ x : int$ を表す。この導出は妥当であるので、term1 は図 2 の構文で定義することができる。

```
term1 : STm (• , Int) Int
term1 = app (lam (var z)) (var z)
```

しかし、上と同じコンテキストのもとで $x @ x : int$ を定義することはできない。以下の term2 を Agda で定義しようとすると、型エラーとなる。なぜなら、app は第 1 引数に Arrow 型の項を受

構文：

$$\begin{aligned} A &= int \mid A \to A \\ \Gamma &= \bullet \mid \Gamma, A \\ t &= x \mid \lambda.\, t \mid t \mathbin{@} t \\ x &= 0 \mid 1 \mid 2 \mid \ldots \end{aligned}$$

型の構成規則：

$$\frac{}{int\ type}\ (Int) \qquad \frac{A\ type \quad B\ type}{A \to B\ type}\ (\to)$$

コンテキストの構成規則：

$$\frac{}{\bullet\ valid}\ (\bullet) \qquad \frac{\Gamma\ valid \quad A\ type}{\Gamma, A\ valid}\ (,)$$

型付け規則：

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ (\text{Var}) \qquad \frac{\Gamma, A \vdash t : B}{\Gamma \vdash \lambda.\, t : A \to B}\ (\text{Lam}) \qquad \frac{\Gamma \vdash t_1 : A \to B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 \mathbin{@} t_2 : B}\ (\text{App})$$

**図 1.** 単純型付き λ 計算

```
data STy : Set where
  Int : STy
  Arrow : STy → STy → STy

data SCon : Set where
  • : SCon
  _,_ : SCon → STy → SCon

data _∋_ : SCon → STy → Set where
  z : {Γ : SCon} {A : STy} → (Γ , A) ∋ A
  s : {Γ : SCon} {A B : STy} → Γ ∋ A → (Γ , B) ∋ A

data STm : SCon → STy → Set where
  var : {Γ : SCon} {A : STy} → Γ ∋ A → STm Γ A
  lam : {Γ : SCon} {A B : STy} → STm (Γ , A) B → STm Γ (Arrow A B)
  app : {Γ : SCon} {A B : STy} → STm Γ (Arrow A B) → STm Γ A → STm Γ B
```

**図 2.** 単純型付き λ 計算の Agda による定義

け取るコンストラクタとして定義されているにもかかわらず、Int 型の項 var z が渡されているからである。

```
term2 : STm (• , Int) Int
term2 = { app (var z) (var z) } -- type error
```

さらに、対象言語における開いた項も定義することができない。term3 は空のコンテキストのもとで $x : int$ を導出しようとしているのに対応するが、これは型エラーとなる。なぜなら、var z が空のコンテキスト • の 0 番目の要素を指そうとしているからである。

```
term3 : STm • Int
term3 = { var z } -- type error
```

このように、図 2 の構文で定義される項は、well-scoped かつ well-typed であることが保証される。

## 3 単純型付き $\lambda$ 計算の CPS 変換

図 1 の構文に対する call-by-name の CPS 変換を考える。call-by-name は、関数適用をする際に、引数を評価せずに関数を適用するという評価戦略のことである。たとえば、$(\lambda.\,0)\mathbin{@}((\lambda.\,1)\mathbin{@}(\lambda.\,1))$ を評価する場合、引数の $(\lambda.\,1)\mathbin{@}(\lambda.\,1)$ がそのまま関数 $\lambda.\,0$ の変数 0 に代入される。

型付きの $\lambda$ 計算に対する CPS 変換は、型に対する変換 $\_^{\div}$、コンテキストに対する変換 $\_^{\div Con}$、項に対する変換 $[\![\_]\!]$ によって、以下のように定義される [7]。ここでは簡単のため、answer type を $int$ に固定する。また、説明のために、変数を de Bruijn index ではなく名前で表現している。

$$
\begin{aligned}
A^{\div} &= (A^{+} \to int) \to int \\
int^{+} &= int \\
(A \to B)^{+} &= A^{\div} \to B^{\div} \\
\bullet^{\div Con} &= \bullet \\
(\Gamma, A)^{\div Con} &= \Gamma^{\div Con}, A^{\div} \\
[\![x]\!] &= x \\
[\![\lambda x.\,t]\!] &= \lambda k.\,k \mathbin{@} (\lambda x.\,[\![t]\!]) \\
[\![t_1 \mathbin{@} t_2]\!] &= \lambda k.\,[\![t_1]\!] \mathbin{@} (\lambda t_1'.\,t_1' \mathbin{@} [\![t_2]\!] \mathbin{@} k)
\end{aligned}
$$

$\_^{\div}$ は型に対するトップレベルの変換、$\_^{+}$ は型の構造に従って再帰するための変換である。コンテキストは、中の型がすべて $\_^{\div}$ で変換されたものとなる。変数 $x$ はそれ自身に変換されるが、型が $A$ から $A^{\div}$ に変化し、継続を受け取ったら、自身をその継続で呼び出すという表現になる。$\lambda$ 抽象は、本体に CPS 変換が適用された $\lambda$ 抽象を自身の継続に渡すような形となる。関数適用 $t_1 \mathbin{@} t_2$ の変換は、$t_1$ を計算し、その値 $t_1'$ を $t_2$ の変換で呼び出した結果を継続 $k$ で実行することを意味する。$t_1'$ に $t_2$ の変換結果がそのまま渡されているため、このように変換された項は call-by-name で評価される。

項の変換を見ると、$\lambda$ 抽象と関数適用のケースでは、CPS 変換によって新たに変数が導入されている。具体的にいうと、前者においては継続を表す変数 $k$、後者においては継続 $k$ および $t_1$ の計算結果 $t_1'$ が導入されている。de Bruijn index を用いた場合、これらの変数の型がコンテキストに追加されるたびに、もとの項に含まれていた変数の index をずらす操作が必要となる。たとえば、$\lambda x.\,x$ を表す項 $\lambda.\,0$ を CPS 変換すると $\lambda.\,(0 \mathbin{@} \lambda.\,1)$ となり、変換前の変数 0 が変換後において 1

構文：

$$\begin{aligned} A &= int \mid A \to_t A \mid A \to_c A \\ \Gamma, \Gamma' &= \bullet \mid \Gamma, A \\ t &= x_t \mid x_c \mid \lambda_t.\, t \mid \lambda_c.\, t \mid t \mathbin{@_t} t \mid t \mathbin{@_c} t \\ x_t &= 0_t \mid 1_t \mid 2_t \mid ... \\ x_c &= 0_c \mid 1_c \mid 2_c \mid ... \end{aligned}$$

型の構成規則：

$$\frac{}{int\ type}\ (Int) \qquad \frac{A\ type \quad B\ type}{A \to_t B\ type}\ (\to_t) \qquad \frac{A\ type \quad B\ type}{A \to_c B\ type}\ (\to_c)$$

コンテキストの構成規則：

$$\frac{}{\bullet\ valid}\ (\bullet) \qquad \frac{\Gamma\ valid \quad A\ type}{\Gamma, A\ valid}\ (,)$$

型付け規則：

$$\frac{\Gamma(x_t) = A}{\Gamma; \Gamma' \vdash x_t : A}\ (\mathrm{Var_t}) \qquad \frac{\Gamma'(x_c) = A}{\Gamma; \Gamma' \vdash x_c : A}\ (\mathrm{Var_c})$$

$$\frac{(\Gamma, A); \Gamma' \vdash t : B}{\Gamma; \Gamma' \vdash \lambda_t.\, t : A \to_t B}\ (\mathrm{Lam_t}) \qquad \frac{\Gamma; (\Gamma', A) \vdash t : B}{\Gamma; \Gamma' \vdash \lambda_c.\, t : A \to_c B}\ (\mathrm{Lam_c})$$

$$\frac{\Gamma; \Gamma' \vdash t_1 : A \to_t B \quad \Gamma; \Gamma' \vdash t_2 : A}{\Gamma; \Gamma' \vdash t_1 \mathbin{@_t} t_2 : B}\ (\mathrm{App_t}) \qquad \frac{\Gamma; \Gamma' \vdash t_1 : A \to_c B \quad \Gamma; \Gamma' \vdash t_2 : A}{\Gamma; \Gamma' \vdash t_1 \mathbin{@_c} t_2 : B}\ (\mathrm{App_c})$$

**図 3.** CPS 変換後の言語

にシフトされる。このように index が変わると、もとの項と変換後の項との対応関係が見づらくなるうえ、変換に関する性質を議論する際に、証明が複雑になってしまう [12]。これは、CPS 変換後の言語を別途定義することによって回避することができる [12, 20]。

図 3 に CPS 変換後の言語を示す。この言語では、変換前の項に含まれる変数の型と、CPS 変換によって導入された変数の型が別々のコンテキストに入れられる。型付け規則を見ると、それぞれの判断が $\Gamma; \Gamma' \vdash t : A$ という形になっており、型付けに 2 つのコンテキスト $\Gamma, \Gamma'$ が使われていることが分かる。もとの項に含まれる変数は $\Gamma$ を参照し、変換後に導入された変数は $\Gamma'$ を参照する。$\lambda$ 抽象は、$\Gamma$ に型を追加する $\lambda_t$ と、$\Gamma'$ に型を追加する $\lambda_c$ の 2 種類が定義され、それぞれ結果の型は $\to_t$ と $\to_c$ になる。関数適用についても、関数部分の型が $\to_t$ のものと $\to_c$ のものを、それぞれ $@_t, @_c$ として定義する。

図 1 の構文から図 3 の構文への CPS 変換は次のように定義される。

$$\begin{aligned} A^{\div} &= (A^{+} \to_c int) \to_c int \\ int^{+} &= int \\ (A \to B)^{+} &= A^{\div} \to_t B^{\div} \\ [\![x]\!] &= x_t \\ [\![\lambda.\, t]\!] &= \lambda_c.\, 0_c \mathbin{@_c} (\lambda_t.\, [\![t]\!]) \\ [\![t_1 \mathbin{@} t_2]\!] &= \lambda_c.\, [\![t_1]\!] \mathbin{@_c} (\lambda_c.\, 0_c \mathbin{@_t} [\![t_2]\!] \mathbin{@_c} 1_c) \end{aligned}$$

なお、変換後の言語における 2 種類のコンテキスト $\Gamma, \Gamma'$ には、どちらにも添字 $t$ が付いた型と $c$ が付いた型の両方が入り得る。たとえば、$A$ 型の変数 0 は $A^{\div} = (A^{+} \to_c int) \to_c int$ 型の変数 $0_t$ に変換されるが、$0_t$ はもとの項に含まれる変数の変換結果であるため、その型は $\Gamma$ に入れられる。一方、$A \to B$ 型の $t_1$ と $A$ 型の $t_2$ の関数適用の変換に含まれる変数 $0_c$ は、$(A \to B)^{+} = A^{\div} \to_t B^{\div}$ 型を持つが、$0_c$ は変換後に導入された変数なので、その型は $\Gamma'$ に入れられる。

上の CPS 変換は、Agda で型付きの項から型付きの項への関数として定義することができる。つまり、変換の前後において型が保存される。

**定理 1** (型の保存性)**.** 任意のコンテキスト $\Gamma$, 項 $t$, 型 $A$ について、$\Gamma \vdash t : A$ が成り立つならば $\Gamma^{\div Con}; \bullet \vdash [\![t]\!] : A^{\div}$ が成り立つ。

## 4 Π 型への拡張

ここからは、Π 型が入った λ 計算を考える。なお、本節は対象言語への Π 型の導入が言語全体にどのような変更をもたらすかを概説することを目的としており、文法などは示さない。本研究で実際に用いた対象言語については、5 節で詳しく述べる。

$\Pi(x : A)B$ は $A \to B$ を一般化したものであり、型 $B$ が項 $x$ に依存することを許す。この型を持つ項は、「$A$ を満たす任意の $x$ について、$B$ が成り立つ」ということの証明となる。

図 1 の関数型 $\to$ を Π 型に置き換えると、型がコンテキストに依存するようになり、型の構成規則が $\Gamma \vdash A\ type$ という形になる。これは、型 $\Pi\ A\ B$ において、$B$ が $A$ 型の自由変数 0 を含むことが許されるためである。図 1 の構文に Π 型を導入しただけでは型の中に変数を書くことができないが、それを許すコンストラクタを型の定義に追加すれば、$B$ の中で変数 0 を使って $A$ 型の項を参照することができる。

また、Π 型を導入すると、関数適用を行った結果の型が複雑になる。$\Pi\ A\ B$ 型の項 $t_1$ と $A$ 型の項 $t_2$ の関数適用は、型 $B[0 \mapsto t_2]$ を持つ。これは、$B$ の中の変数 0 を $t_2$ に置き換えたような型を表す。単純型付き λ 計算の関数適用は、評価する際に項レベルの代入を伴うが、Π 型が入ると、型を付ける際に型レベルの代入が必要になる。

## 5 Agda による Π 型付き λ 計算の定義

本節では、Altenkirch and Kaposi [2] に従って、Π 型を含む言語を Agda で定義する。構文を図 4 に、Agda による実装を図 5 に示す。Agda の型定義を見ると、コンテキスト、型、項のほかに、代入 Tms が追加されている。単純型付き λ 計算のときと異なり、これらの定義は互いに依存している。たとえば、Ty の要素の 1 つである Π の定義に注目すると、2 つ目の引数の型にコンテキストのコンストラクタ , が使われている。そのため、先頭に mutual というキーワードを付けて、これらのデータ型を相互再帰的に定義している。それぞれのデータ型については以下で解説する。

### 5.1 コンテキストと型

コンテキストは型の列であるが、Γ, A というコンテキストにおいて、A は Γ のもとで構成された型であることが要求される。型はコンテキストで index されている。U は単純型付き λ 計算における int と同じようにベースとなる型であり、同時に kind でもある。El は U 型の項を型レベルに持ち上げるためのコンストラクタである。これらと Π のほかに、型に対して代入を行うためのコンストラクタ _[_]T が定義されている（_ は引数の位置を表す）。

構文：

$$\begin{array}{rcl} \Gamma & = & \bullet \mid \Gamma, A \\ A & = & U \mid El\ t \mid \Pi\ A\ A \mid A[\delta]_T \\ \delta & = & \epsilon \mid \delta, t \mid id \mid \delta \circ \delta \mid \pi_1\ \delta \\ t & = & \lambda.t \mid app\ t \mid \pi_2\ \delta \mid t[\delta]_t \end{array}$$

コンテキストの構成規則：

$$\frac{}{\bullet\ valid}\ (\bullet) \qquad \frac{\Gamma\ valid \quad \Gamma \vdash A\ type}{\Gamma, A\ valid}\ (,)$$

型の構成規則：

$$\frac{\Gamma\ valid}{\Gamma \vdash U\ type}\ (\mathrm{U}) \qquad \frac{\Gamma \vdash t : U}{\Gamma \vdash El\ t\ type}\ (\mathrm{El})$$

$$\frac{\Gamma \vdash A\ type \quad \Gamma, A \vdash B\ type}{\Gamma \vdash \Pi\ A\ B\ type}\ (\Pi) \qquad \frac{\Delta \vdash A\ type \quad \delta : \Delta \Rightarrow \Gamma}{\Gamma \vdash A[\delta]_T\ type}\ ([]_{\mathrm{T}})$$

代入の構成規則：

$$\frac{\Gamma\ valid}{\epsilon : \bullet \Rightarrow \Gamma}\ (\epsilon) \qquad \frac{\delta : \Delta \Rightarrow \Gamma \quad \Gamma \vdash t : A[\delta]_T}{\delta, t : \Delta, A \Rightarrow \Gamma}\ (,)$$

$$\frac{\Gamma\ valid}{id : \Gamma \Rightarrow \Gamma}\ (\mathrm{Id}) \qquad \frac{\delta : \Sigma \Rightarrow \Delta \quad \sigma : \Delta \Rightarrow \Gamma}{\delta \circ \sigma : \Sigma \Rightarrow \Gamma}\ (\circ) \qquad \frac{\delta : \Delta, A \Rightarrow \Gamma}{\pi_1\ \delta : \Delta \Rightarrow \Gamma}(\pi_1)$$

型付け規則：

$$\frac{\Gamma, A \vdash t : B}{\Gamma \vdash \lambda.t : \Pi\ A\ B}\ (\mathrm{Lam}) \qquad \frac{\Gamma \vdash t : \Pi\ A\ B}{\Gamma, A \vdash app\ t : B}\ (\mathrm{App})$$

$$\frac{\delta : \Delta, A \Rightarrow \Gamma}{\Gamma \vdash \pi_2\ \delta : A[\pi_1\ \delta]_T}\ (\pi_2) \qquad \frac{\Delta \vdash t : A \quad \delta : \Delta \Rightarrow \Gamma}{\Gamma \vdash t[\delta]_t : A[\delta]_T}\ ([]_{\mathrm{t}})$$

**図 4.** Π 型付き λ 計算

```
mutual
  data Con : Set where
    • : Con
    _,_ : (Γ : Con) → Ty Γ → Con

  data Ty : Con → Set where
    U : {Γ : Con} → Ty Γ
    El : {Γ : Con} → (A : Tm Γ U) → Ty Γ
    Π : {Γ : Con} → (A : Ty Γ) → (B : Ty (Γ , A)) → Ty Γ
    _[_]T : {Γ Δ : Con} → Ty Δ → Tms Γ Δ → Ty Γ

  data Tms : Con → Con → Set where
    ε : {Γ : Con} → Tms Γ •
    _,_ : {Γ Δ : Con} {A : Ty Δ} →
      (δ : Tms Γ Δ) → Tm Γ (A [ δ ]T) → Tms Γ (Δ , A)
    id : {Γ : Con} → Tms Γ Γ
    _∘_ : {Γ Δ Σ : Con} → Tms Δ Σ → Tms Γ Δ → Tms Γ Σ
    π₁ : {Γ Δ : Con} {A : Ty Δ} → Tms Γ (Δ , A) → Tms Γ Δ

  data Tm : (Γ : Con) → Ty Γ → Set where
    lam : {Γ : Con} {A : Ty Γ} {B : Ty (Γ , A)} →
      Tm (Γ , A) B → Tm Γ (Π A B)
    app : {Γ : Con} {A : Ty Γ} {B : Ty (Γ , A)} →
      Tm Γ (Π A B) → Tm (Γ , A) B
    π₂ : {Γ Δ : Con} {A : Ty Δ} →
      (δ : Tms Γ (Δ , A)) → Tm Γ (A [ π₁ δ ]T)
    _[_]t : {Γ Δ : Con} {A : Ty Δ} →
      Tm Δ A → (δ : Tms Γ Δ) → Tm Γ (A [ δ ]T)
```

**図 5.** Π 型付き λ 計算の Agda による定義

## 5.2 代入

Altenkirch らの体系において、代入は明示的なコンストラクタとして表されている。Tms Γ Δ は、Δ のもとで構成された型（項）を Γ のもとで構成された型（項）にする操作を表す（図 4 では Δ ⇒ Γ と表現している）。直感的に、この代入は Γ のもとで型がついた項の列として理解することができる。これらの項は変数によって参照される。変数の表現については後述するが、$n$ 番目の変数は代入の $n$ 番目の要素を指す。こうして、もともと Δ で index された型（項）の中の変数を Γ で index された項に置き換えることで、全体として Γ で index された型（項）になるのである。代入の各コンストラクタは次のように解釈される。$\epsilon$ は空の代入を表す。_, _ は代入を拡張するためのコンストラクタである。結果の型において、代入前のコンテキストが 1 つ長くなっているのが、拡張されたことを表している。id は何もしない代入、_ ∘ _ は代入どうしの合成である。$\pi_1$ は _, _ の逆で、項の列を 1 つ短くする。

なお、この体系において、代入は「どの変数をどの項で置き換えるか」という情報を示すものであり、変数から項への置き換えは実際に行われない。これは、通常の $\beta$ 簡約の規則に現れる代入の扱いとは異なる。$\beta$ 簡約の規則 $(\lambda x.\, e_1)\ e_2 \to_\beta e_1[e_2/x]$ [6] において、代入 $[e_2/x]$ はメタレベルの計算として実行される。Altenkirch らがこのような実際の置き換えを伴う代入を採用していないのは、代入をコンストラクタとして扱う体系は実装が比較的単純であるうえ、categories with families [14] とよばれる圏との対応がつきやすくなるためである。

```
postulate
  [id]T : {Γ : Con} {A : Ty Γ} → A [ id ]T ≡ A
  []T : {Γ Δ Σ : Con} {A : Ty Σ} → (δ : Tms Δ Σ) →
    (σ : Tms Γ Δ) → A [ δ ]T [ σ ]T ≡ A [ δ ∘ σ ]T
  U[] : {Γ Δ : Con} {δ : Tms Γ Δ} → U [ δ ]T ≡ U
  El[] : {Γ Δ : Con} {A : Tm Δ U} {δ : Tms Γ Δ} →
    El A [ δ ]T ≡ El (coe (Tm Γ ≡ U[]) (A [ δ ]t))
```

**図 6.** 等価関係の定義

### 5.3 Postulate による等価関係の定義

代入をコンストラクタとして表すと、非自明な等価関係が生じる。たとえば、型 A に何もしない代入を適用した型 A [ id ]T は、直感的に A と等しいが、コンストラクタ _[_]T が余計に付いているため、Agda はこれを A と異なるものとして認識する。 そのため、A [ id ]T 型の項が要求されているところに A 型の項を書くと、型エラーが生じてしまう。A と A [ id ]T は異なる方法で構成された型であるため、その間の等価性を証明することは不可能である。そこで、このような等価関係を postulate というキーワードを使って定義する。このキーワードを使うと、証明を与えずに、型を宣言するだけで項を定義することができる。図 6 に等価関係の例を示す。Altenkirch らは、項と代入に関する等価関係なども定義している。これらの等価関係は、代入の意味を定義しているとみなすことができる。

Postulate によって定義された等価関係を使って項の型を書き換えるときには、以下の 2 つの関数を用いる。TmΓ ≡ は、2 つの型 $A_0$, $A_1$ が等しいなら、Tm Γ $A_0$ と Tm Γ $A_1$ を同じ型とみなす、ということを表す。coe は、型 A, B が等しいなら、A 型の項を B 型の項にすることができる、ということを意味する。

```
Tm Γ ≡ : {Γ : Con} {A₀ A₁ : Ty Γ} → (A₂ : A₀ ≡ A₁) →
  Tm Γ A₀ ≡ Tm Γ A₁
Tm Γ ≡ {Γ} refl = refl

coe : ∀ {i} {A B : Set i} → A ≡ B → A → B
coe refl a = a
```

### 5.4 項

項 Tm Γ A は、単純型付き λ 計算のときと同じように、Γ のもとで A 型を持つ項を表すが、A が Γ によって index された型であることを要求する。lam は λ 抽象に対応するコンストラクタであり、app はその双対である（後者については 5.6 節で述べる）。$\pi_2$ は代入の先頭の項を取り出す操作を表す。_[_]t は項に対して代入を行うためのコンストラクタである。

### 5.5 変数

図 5 の構文には変数に対応するコンストラクタがないが、変数は代入を使って以下のように定義することができる。

```
wk : {Γ : Con} {A : Ty Γ} → Tms (Γ , A) Γ
wk = π 1 id

vz : {Γ : Con} {A : Ty Γ} → Tm (Γ , A) (A [ wk ]T)
vz = π 2 id

vs : {Γ : Con} {A B : Ty Γ} → Tm Γ A → Tm (Γ , B) (A [ wk ]T)
vs x = x [ wk ]t
```

一般に、変数は vs (vs (... vz)) と表される。0 番目の変数 vz は、$\pi_2$ を使って空でない代入の先頭要素を参照する。1 番目以降の変数は、1 つ短いコンテキストのもとでの表現に弱化 wk を適用することで表現される。$\pi_2$ と _[_]t の定義より、変数は必ず A [ wk ]T という形の型を持つ。wk はコンテキストに余分な要素を 1 つ加える操作に対応する。変数の表現に wk が必要なのは、変数の型を含んでいるコンテキストと、その型を index しているコンテキストの長さを合わせなければならないからである。コンテキストに入っている変数の型は、それぞれ自身の直前までのコンテキストから導出される。たとえば、$(\Gamma, A), B$ の A は $\Gamma$ から導出される。このコンテキストに B が付け加わると、A の型も 1 つ長くなったコンテキストで index されなければならない。このために wk を適用しているのである。

### 5.6 関数適用

図 5 の app コンストラクタは、lam の双対として定義されており、2 節で見たような 2 項を受け取るコンストラクタと異なる形をしている。しかし、app と代入を用いると、通常の関数定義を次のように定義することができる。

```
< _ > : {Γ : Con} {A : Ty Γ} → Tm Γ A → Tms Γ (Γ , A)
< t > = id , coe (Tm Γ ≡ ([id]T ⁻¹)) t

_$_ : {Γ : Con} {A : Ty Γ} {B : Ty (Γ , A)} →
  (t₁ : Tm Γ (Π A B)) → (t₂ : Tm Γ A) → Tm Γ (B [ < t₂ > ]T)
t₁ $ t₂ = (app t₁) [ < t₂ > ]t
```

4 節で述べたように、$\Pi\ A\ B$ 型の関数 $t_1$ を $t_2$ に適用すると、結果の型は $B$ に「0 番目の変数を $t_2$ に置き換える」という代入が付いたものになる。上の定義において、$t_1$ は Π A B 型の関数であり、これを app コンストラクタに渡すと、全体として「0 番目の変数が A 型であるような、B 型の項」となる。これに対して、$< t_2 >$ という代入が適用されている。この代入は、だいたい「0 番目の変数を $t_2$ に置き換える」という操作に対応しているが、「型 A は、自身に何もしない代入をかけた型と等しい」という等価関係を使って、$t_2$ の型を A 型から A [ id ]T 型に書き換えている（$\_^{-1}$ は $A \equiv B$ を $B \equiv A$ にする関数である）。これが必要なのは、代入 $\delta$ , t の定義より、項 t に代入 $\delta$ が適用されていることが要求されるからである。

### 5.7 項の例

上の構文で表現できる項の例を見てみよう。以下の term1 は、恒等関数 $\lambda x : U.\, x$ を表す。

```
term1 : {Γ : Con} → Tm Γ (Π U U)
term1 = lam (coe (Tm Γ ≡ U[]) vz)
```

coe を使って変数 vz の型を書き換えているのに注目しよう。term1 全体の型が Π U U であるため、vz は U 型を要求されているが、変数の定義より、vz は U [ wk ]T 型を持つ。そこで、U に代入が付いた型は U と等しいことを表す等価関係 U[] を使って、[ wk ]T を除去している。

term1 は単純型付き $\lambda$ 計算でも定義することが可能である。一方、コンストラクタ El を使うと、項に依存した型を書くことができる。以下の term2 は polymorphic な恒等関数 $\lambda t : U.\, \lambda x : t.\, x$ を表す。変数 $x$ の型が項 $t$ となっているため、この項は単純型付き $\lambda$ 計算で表現することができない。

```
term2 : Tm • (Π U
  (Π (El (coe (Tm Γ ≡ U[]) vz))
    (El (coe (Tm Γ ≡ U[]) ((coe (Tm Γ ≡ U[]) vz) [ wk ]t)))))
term2 = lam (lam (coe (Tm Γ ≡ El[]) vz))
```

型の 2 行目を見ると、$x$ の型は El (coe (TmΓ ≡ U[]) vz) となっている。$x$ は型 $t$ を持つが、$t$ は項であるため、El を使って型レベルに持ち上げている。$t$ は直前の Π で導入された変数なので、vz で指すことができるが、変数の定義より、vz の型は U に弱化が適用された U [ wk ]T となっている。El に渡すことができるのは U 型の項のみであるため、U[] を使って vz の型を U [ wk ]T から U に書き換えている。3 行目は $x$ の型を表す。$x$ の型は $t$、すなわち coe (TmΓ ≡ U[]) vz だが、2 つ目の Π によってコンテキストが 1 つ長くなっているため、弱化 wk を適用する必要がある。_[]t の定義より、項に適用された代入は型にも適用されるため、vz [ wk ]t は型 U [ wk ]T を持つ。そこで、U[] を使ってこの項の型を U に書き換え、El に渡している。一方、項の定義では、$x$ に相当する項が coe (TmΓ ≡ El[]) vz となっている。$x$ は最も内側の lam で束縛された項なので、vz で表されるが、その型は 2 つ目の Π の第 1 引数に [ wk ]T が付いたものとなる。そこで、El[] という等価関係を使う。この等価関係は、「El A に代入がついた型は、項 A にその代入を適用し、U[] で型を書き換えた結果を El に渡したような型と等しい」ということを表す。これを使って vz の型を書き換えると、vz の型は求められている型（型の 3 行目）と一致する。

次の term3 は、型の中に関数適用を持つ項、つまり、依存型が入った体系でしか表現することができない項の例である。この項は term2 の $x$ の型を $(\lambda t' : U.\, t')\ t$ という項に置き換えたものであり、全体として $\lambda t : U.\, \lambda x : (\lambda t' : U.\, t')\ t.\, x$ を表す。項の定義を見ると、term2 の型の中に現れる vz が、$(\lambda t' : U.\, t')\ t$ を表す term1 $ coe (Tm Γ ≡ U[]) vz に置き換わっていることが分かる。

```
term3 : Tm • (Π U
  (Π (El (coe (Tm Γ ≡ U[]) (term1 $ coe (Tm Γ ≡ U[]) vz)))
    (El (coe (Tm Γ ≡ U[]) (coe (Tm Γ ≡ U[]) (term1 $ coe (Tm Γ ≡ U[]) vz) [ wk ]t)))))
term3 = lam (lam (coe (Tm Γ ≡ El[]) vz))
```

## 6 Π 型付き λ 計算の CPS 変換

本節では、5 節の構文に対する call-by-name の CPS 変換を考える。なお、評価戦略として call-by-name を選んでいるのは、Π 型を含む体系において call-by-value の CPS 変換を定義しようとすると、関数適用のケースで型が付かなくなるためである [20]。

まず、3 節と同様に、CPS 変換後の言語を図 7 のように定義する。Agda において、コンテキスト、型、代入、項の型定義はそれぞれ以下のようになる。

```
data CCon : Set
data CTy : CCon → CCon → Set
data CTms : CCon → CCon → CCon → CCon → Set
data CTm : (Γ Γ ' : CCon) → CTy Γ Γ ' → Set
```

型と項は 2 つのコンテキストによって index される。このうち、1 つ目は変換前の項に含まれる変数の型が入ったもの、2 つ目は変換後に導入された変数の型が入ったものである。代入は 4 つのコンテキストを受け取る。最初の 2 つは代入後における 2 種類のコンテキスト、後の二つは代入後における 2 種類のコンテキストを表す。すなわち、Tms Γ Γ′ Δ Δ′ は、Δ と Δ′ によって index された型（項）を Γ と Γ′ によって index された型（項）にするような代入を表す。図 7 において、このような代入は $\Delta; \Delta' \Rightarrow \Gamma; \Gamma'$ と表されている。

次に、図 5 の Tm から上の CTm への CPS 変換を図 8 のように定義する。$\_^{\div Con}$, $\_^{\div}$, $\_^{\div Tms}$, $[\![\_]\!]$ は、それぞれコンテキスト、型、代入、項に対する変換である。Answer type は $U$ とする。また、$lam\ t$, $app\ t$ の変換における $wkt$ と $wkc$ は、CPS 変換後の言語における 2 種類のコンテキストをそれぞれ 1 つ拡張するような弱化を表す。

Agda において、コンテキスト、型、代入、項に対する変換は、それぞれ次のような型を持つ。

構文：

$$\begin{array}{rcl}\Gamma & = & \bullet \mid \Gamma,_t\, A \\ \Gamma' & = & \bullet \mid \Gamma',_c\, A \\ A & = & U \mid El\ t \mid \Pi_t\ A\ A \mid \Pi_c\ A\ A \mid A[\delta]_T \\ \delta & = & \epsilon \mid \delta,_t\, t \mid \delta,_c\, t \mid id \mid \delta \circ \delta \mid \pi_{1t}\ \delta \mid \pi_{1c}\ \delta \\ t & = & \lambda_t.\, t \mid \lambda_c.\, t \mid app_t\ t \mid app_c\ t \mid \pi_{2t}\ \delta \mid \pi_{2c}\ \delta \mid t[\delta]_t\end{array}$$

コンテキストの構成規則：

$$\frac{}{\bullet\ valid}\ (\bullet) \qquad \frac{\Gamma\ valid \quad \Gamma;\Gamma' \vdash A\ type}{\Gamma,_t\, A\ valid}\ (,_t) \qquad \frac{\Gamma'\ valid \quad \Gamma;\Gamma' \vdash A\ type}{\Gamma',_c\, A\ valid}\ (,_c)$$

型の構成規則：

$$\frac{\Gamma,\Gamma'\ valid}{\Gamma;\Gamma' \vdash U\ type}\ (\mathrm{U}) \qquad \frac{\Gamma;\Gamma' \vdash t : U}{\Gamma;\Gamma' \vdash El\ t\ type}\ (\mathrm{El}) \qquad \frac{\Delta;\Delta' \vdash A\ type \quad \delta : \Delta;\Delta' \Rightarrow \Gamma;\Gamma'}{\Gamma;\Gamma' \vdash A[\delta]_T\ type}\ ([]_{\mathrm{T}})$$

$$\frac{\Gamma;\Gamma' \vdash A\ type \quad (\Gamma,_t\, A);\Gamma' \vdash B\ type}{\Gamma;\Gamma' \vdash \Pi_t\ A\ B\ type}\ (\Pi_t) \qquad \frac{\Gamma;\Gamma' \vdash A\ type \quad \Gamma;(\Gamma',_c\, A) \vdash B\ type}{\Gamma;\Gamma' \vdash \Pi_c\ A\ B\ type}\ (\Pi_c)$$

代入の構成規則：

$$\frac{\Gamma,\Gamma'\ valid}{\epsilon : \bullet;\bullet \Rightarrow \Gamma;\Gamma'}\ (\epsilon)$$

$$\frac{\delta : \Delta;\Delta' \Rightarrow \Gamma;\Gamma' \quad \Gamma;\Gamma' \vdash t : A[\delta]_T}{\delta,_t\, t : (\Delta,_t\, A);\Delta' \Rightarrow \Gamma;\Gamma'}\ (,_t) \qquad \frac{\delta : \Delta;\Delta' \Rightarrow \Gamma;\Gamma' \quad \Gamma;\Gamma' \vdash t : A[\delta]_T}{\delta,_c\, t : \Delta;(\Delta',_c\, A) \Rightarrow \Gamma;\Gamma'}\ (,_c)$$

$$\frac{\Gamma,\Gamma'\ valid}{id : \Gamma;\Gamma' \Rightarrow \Gamma;\Gamma'}\ (\mathrm{Id}) \qquad \frac{\delta : \Sigma;\Sigma' \Rightarrow \Delta;\Delta' \quad \sigma : \Delta;\Delta' \Rightarrow \Gamma;\Gamma'}{\delta \circ \sigma : \Sigma;\Sigma' \Rightarrow \Gamma;\Gamma'}\ (\circ)$$

$$\frac{\delta : (\Delta,_t\, A);\Delta' \Rightarrow \Gamma;\Gamma'}{\pi_{1t}\ \delta : \Delta;\Delta' \Rightarrow \Gamma;\Gamma'}(\pi_{1t}) \qquad \frac{\delta : \Delta;(\Delta',_c\, A) \Rightarrow \Gamma;\Gamma'}{\pi_{1c}\ \delta : \Delta;\Delta' \Rightarrow \Gamma;\Gamma'}(\pi_{1c})$$

型付け規則：

$$\frac{(\Gamma, A);\Gamma' \vdash t : B}{\Gamma;\Gamma' \vdash \lambda_t\, t : \Pi_t\ A\ B}\ (\mathrm{Lam_t}) \qquad \frac{\Gamma;(\Gamma', A) \vdash t : B}{\Gamma;\Gamma' \vdash \lambda_c\, t : \Pi_c\ A\ B}\ (\mathrm{Lam_c})$$

$$\frac{\Gamma;\Gamma' \vdash t : \Pi\ A\ B}{(\Gamma,_t\, A);\Gamma' \vdash app_t\ t : B}\ (\mathrm{App_t}) \qquad \frac{\Gamma;\Gamma' \vdash t : \Pi_c\ A\ B}{\Gamma;(\Gamma',_c\, A) \vdash app_c\ t : B}\ (\mathrm{App_c})$$

$$\frac{\delta : (\Delta,_t\, A);\Gamma' \Rightarrow \Gamma;\Gamma'}{\Gamma;\Gamma' \vdash \pi_{2t}\ \delta : A[\pi_{1t}\ \delta]_T}\ (\pi_{2t}) \qquad \frac{\delta : \Gamma;(\Delta',_c\, A) \Rightarrow \Gamma;\Gamma'}{\Gamma;\Gamma' \vdash \pi_{2c}\ \delta : A[\pi_{1c}\ \delta]_T}\ (\pi_{2c})$$

$$\frac{\Delta;\Delta' \vdash t : A \quad \delta : \Delta;\Delta' \Rightarrow \Gamma;\Gamma'}{\Gamma;\Gamma' \vdash t[\delta]_t : A[\delta]_T}\ ([]_{\mathrm{t}})$$

**図 7.** CPS 変換後の言語

$$
\begin{aligned}
\bullet^{\div Con} &= \bullet \\
(\Gamma, A)^{\div Con} &= \Gamma^{\div Con}{}_{,t}\, A^{\div} \\
A^{\div} &= \Pi_c\ (\Pi_c\ A^{+}\ U)\ U \\
U^{+} &= U \\
(El\ t)^{+} &= El\ (\llbracket t \rrbracket\ \$_c\ (\lambda_c.\, 0_c)) \\
(\Pi\ A\ B)^{+} &= \Pi_t\ A^{\div}\ B^{\div} \\
(A[\delta]_T)^{+} &= A^{+}[\delta^{\div Tms}]_T \\
\epsilon^{\div Tms} &= \epsilon \\
(\delta, t)^{\div Tms} &= \delta^{\div Tms}{}_{,t}\, \llbracket t \rrbracket \\
id^{\div Tms} &= id \\
(\delta \circ \sigma)^{\div Tms} &= \delta^{\div Tms} \circ \sigma^{\div Tms} \\
(\pi_1\ \delta)^{\div Tms} &= \pi_{1t}\ \delta^{\div Tms} \\
\llbracket \lambda.\, t \rrbracket &= \lambda_c.\, 0_c\ \$_c\ ((\lambda_t.\, \llbracket t \rrbracket)[wkc]_t) \\
\llbracket app\ t \rrbracket &= \lambda_c.\, \llbracket t \rrbracket[wkt \circ wkc]_t\ \$_c\ (\lambda_c.\, 0_c\ \$_t\ (0_t[wkc \circ wkc]_t)\ \$_c\ 1_c) \\
\llbracket \pi_2\ \delta \rrbracket &= \pi_{2t}\ \delta^{\div Tms} \\
\llbracket t[\delta]_t \rrbracket &= \llbracket t \rrbracket[\delta^{\div Tms}]_t
\end{aligned}
$$

**図 8.** Π 型付き λ 計算の CPS 変換

```
÷-Con : Con → CCon
÷ : {Γ : Con} → Ty Γ → CTy (÷-Con Γ) •
÷-Tms : {Γ Δ : Con} → Tms Γ Δ → CTms (÷-Con Γ) • (÷-Con Δ) •
cps : {Γ : Con} {A : Ty Γ} → Tm Γ A → CTm (÷-Con Γ) • (÷ A)
```

Agda で図 8 の変換を実装する際には、いくつかのケースにおいて、postulate によって定義された等価関係を使って型を書き換える必要がある。例として、$A[\delta]_T$ 型を持つ項 $t[\delta]_t$ の変換を考える。$\llbracket \_ \rrbracket$ の定義より、この項は $\llbracket t \rrbracket[\delta^{\div Tms}]_t$ に変換される。変換後の項は型 $(\Pi\ (\Pi\ A^{+}\ U)\ U)[\delta^{\div Tms}]_T$ を持つが、もともとの型が $A[\delta]_T$ であったため、変換後の型として求められているのは $(A[\delta]_T)^{\div}$、すなわち $\Pi\ (\Pi\ (A^{+}[\delta^{\div Tms}]_T)\ U)\ U$ である。ここで、等価関係を使って $\Pi\ A\ B$ 全体に適用されている代入を $A$ と $B$ にそれぞれ適用し、さらに $U$ に適用された代入を除去すれば、変換後の項の型を求められた型に書き換えることができる。図 8 では、見やすさを優先するためにこのような型の書き換えを省略している。

図 8 のうち、$El$ の変換に注目しよう。$El\ t$ は、$t$ の変換結果 $\llbracket t \rrbracket$ に継続として恒等関数を渡したものを引数とする $El$ 型に変換される。$El$ の定義より、$t$ は型 $U$ を持ち、$\llbracket t \rrbracket$ は型 $\Pi_c\ (\Pi_c\ U\ U)\ U$ を持つ。Π 型の項は $El$ の引数になり得ないため、$\llbracket t \rrbracket$ を $\Pi_c\ U\ U$ 型の恒等関数 $\lambda_c.\, c_0$ に適用し、$U$ 型の項を作ったうえで、$El$ に渡しているのである。ここで恒等関数を渡しているのは、単に型を合わせているだけではなく、CPS 変換を行う際に、型は継続を受け取る形に変換されないことを表している、と捉えることもできる。

次に、$app\ t$ の変換を考える。この表現は、単純型付き λ 計算における関数適用 $t_1 \mathbin{@} t_2$ と異なる形をしているが、両者の CPS 変換は類似している。

$$
\begin{aligned}
\llbracket t_1 \mathbin{@} t_2 \rrbracket &= \lambda_c.\, \llbracket t_1 \rrbracket \mathbin{@}_c\ (\lambda_c.\, 0_c \mathbin{@}_t \llbracket t_2 \rrbracket \mathbin{@}_c\ 1_c) \\
\llbracket app\ t \rrbracket &= \lambda_c.\, \llbracket t \rrbracket[wkt \circ wkc]_t\ \$_c\ (\lambda_c.\, 0_c\ \$_t\ (0_t[wkc \circ wkc]_t)\ \$_c\ 1_c)
\end{aligned}
$$

上の 2 つの変換は、2 点において異なる。1 つ目は、$t_1 \mathbin{@} t_2$ の変換における $[\![t_1]\!]$ が、$app\ t$ の変換においては $[\![t]\!][wkt \circ wkc]_t$ となっている点である。今、$app\ t$ が $\Gamma, A$ のもとで $B$ 型を持つとすると、$t$ は $\Gamma$ のもとで $\Pi\ A\ B$ 型を持つ。Agda における cps の型を見ると、$app\ t$ の変換結果は $(\Gamma, A)^{\div Con}$ と $\bullet$ で、$t$ の変換結果は $\Gamma^{\div Con}$ と $\bullet$ で index される。したがって、1 つ目のコンテキストの長さを合わせるために、$[\![t]\!]$ に弱化 $wkt$ を適用する必要がある。さらに、$[\![t]\!]$ は $\lambda_c$ の内側に現れているため、2 つ目のコンテキストも 1 つ拡張しなければならない。そのために $wkc$ を適用している。

2 つ目の相違点は、$t_1 \mathbin{@} t_2$ の変換における $[\![t_2]\!]$ が、$app\ t$ の変換では $0_t[wkc \circ wkc]_t$ となっている点である。変数 $0_t$ は、「何か代入が適用されたら、その 0 番目の項を指す」ということを表す。$ の定義より、通常の関数適用 $t_1\ \$\ t_2$ はだいたい $(app\ t_1)[id, t_2]_t$ という形に展開される（実際には、等価関係を使って $t_2$ の型を書き換える必要がある）。つまり、通常の関数適用は $t[\delta]_t$ の特別なケースである。項 $t[\delta]_t$ は $[\![t]\!][\delta^{\div Tms}]_t$ と変換されるため、$app\ t_1$ の変換結果に含まれる $0_t$ は $(id, t_2)^{\div Tms}$ の先頭要素、すなわち $[\![t_2]\!]$ を指す。$0_t$ に $t_2$ の変換結果がそのまま代入されるため、この変換は call-by-name であるといえる。ここで、$\_^{\div Tms}$ の定義より、$[\![t_2]\!]$ を index している 2 つ目のコンテキストは $\bullet$ である。一方、$0_t$ が現れるのは 2 つの $\lambda_c$ のスコープ内である。そのため、$[\![t_2]\!]$ に $wkc$ を 2 回適用し、2 つ目のコンテキストの長さを合わせている（$\lambda.t$ の変換で、$\lambda_t.[\![t]\!]$ に $wkc$ が適用されているのも、同じ理由による）。

現在、著者らは図 8 の変換を Agda で実装しており、app のケースを除けば、すべての変換において期待された型の項が得られることが分かっている。一方、cps のプログラムに app の変換を入れると、型検査が終了しなくなってしまう。その原因が app の変換にあるのか、あるいは Agda の型検査にあるのかは不明であるが、cps が型検査に通れば、型の保存性が示されたことになる。

現在のプログラムがカバーしている項の例として、5.7 節の term1 を CPS 変換してみよう。以下に term1 を再掲する。

```
term1 : {Γ : Con} → Tm Γ (Π U U)
term1 = lam (coe (Tm Γ ≡ U[]) vz)
```

この項を cps に渡した結果は以下の通りである。ただし、Πc[] による型の書き換えは、Πc A B の外側に適用されている代入を A と B に適用するという操作を表す。

```
lamc
 (coe
   (CTmΓ ≡ U[])
     (appc
       (coe
         (CTmΓ ≡
           (trans (Πc[] wkc)
             (trans (cong (Πc (Πt (Πc (Πc U U) U) (Πc (Πc U U) U)) [ wkc ]T)) U[])
                    refl)))
         vzc)
     [ id ,c
       coe (CTmΓ ≡ ([id]T ⁻¹))
           (lamt (cps (coe (TmΓ ≡ U[]) vz)) [ wkc ]t) ]t))
```

上の項は、$\lambda_c.0_c\ \$_c\ ((\lambda_t.[\![0]\!])[wkc]_t)$ に必要な型の書き換えを行ったものである。10 行目の vzc は継続 $0_c$ を表し、2 行目から 9 行目でその型を書き換えている。その下の代入は $\$_c$ に伴うものである。この項全体の型は

```
CTm • • (Πc (Πc (Πt (Πc (Πc U U) U) (Πc (Πc U U) U)) U) U)
```

であり、もとの項の型 Π U U に ÷ を適用したような型で index されている。

5.7 節では、例として多相の恒等関数 $\lambda t : U.\, \lambda x : t.\, x$ も取り上げたが、この項を CPS 変換したところ、結果が返ってこなかった。この項は型に El を含んでいるという点において、term1 より複雑になっている。そこで、型のみを ÷ で変換したところ、これも結果が得られなかった。一方で、これより少し単純な型 El (coe (TmΓ ≡ U[]) vz) は、÷ によって正しく変換された。この問題については、関数適用の型検査と同様に、さらなる考察が必要である。

# 7 関連研究

Π 型が入った体系での CPS 変換は、Barthe et al. [7] によって議論されている。1 節でも述べたように、Barthe らは kind を独立したカテゴリとして定義している。また、項レベルだけでなく、型レベルにおいても変数、λ 抽象、関数適用をそれぞれ定義しているため、El のようなコンストラクタを必要としない。Barthe らは、このような体系において、型を保存する call-by-name の CPS 変換を定義できることを証明しているが、その実装は示していない。なお、Barthe らの定義では、項のみが CPS 変換され、型は継続を受け取る形に変換されない。これは、El の引数を CPS 変換した結果に恒等関数を渡すという発想と対応している。

Barthe and Uustalu [8] は、Π 型と同じ方法で Σ 型に対する CPS 変換を定義しようとすると、型が保存されなくなることを証明している。$\Sigma(x : A)B$ は、ペアの一般形を表す型である。項 $(x, u)$ がこの型を持つとき、$x$ は $A$ 型の項であり、$u$ は $x$ が $B(x)$ を満たすことの証明である。つまり、この型全体として、「$B$ を満たすような $A$ 型の $x$ が存在する」ということを表す。項 $(x, u)$ の第 1 要素と第 2 要素は、それぞれ fst, snd を使って取り出すことができる。このうち、snd の CPS 変換において、継続が要求する項の型と、実際に継続が受け取る項の型が一致しなくなってしまう。

Σ 型を含む体系で継続を扱うことの難しさは、Herbelin [16] でも述べられている。Herbelin は、Σ 型と equality が入った体系に継続演算子 call/cc [10] を導入すると、矛盾することを示している。

# 8 まとめと今後の課題

本研究では、Π 型が入った λ 計算に対する CPS 変換の Agda による実装を試みた。変換前後の項を型付きで表現するために、Altenkirch and Kaposi [2] の構文を採用し、Π 型、$El$ 型、明示的な代入を含む言語に対して CPS 変換を定義した。定義した CPS 変換を Agda で実装したところ、関数適用の変換で型検査が終了しなかったほか、$El$ を含む型の変換が得られない場合があったが、それ以外のケースについては、型が保存されるような CPS 変換が定義可能であることが示された。

今回の実装には未完成の部分が残っているので、型検査や実行が終了しない場合について、その理由を明らかにしたい。また、依存型を含む体系において、コンテキストを 2 つに分けるというアプローチが安全であるか、という議論も必要であると考えられる。Barthe らの CPS 変換の実装も今後の課題の 1 つである。これを行うためには、Barthe らの言語に対して型付きの表現を与えなくてはならない。特に、代入については、本稿で用いた体系では明示的なコンストラクタとして扱っているのに対し、Barthe らはメタレベルの非明示的な演算として扱っているため、それぞれの方法に従って実装し、いずれの場合も実装できるのか、また、どちらがより実装しやすいのかを考察したい。長期的には、対象言語を帰納的データ型などで拡張し、どのような体系で CPS 変換が定義可能であるかを追求したいと考えている。

## 謝辞

ミスの指摘や変換のアイディアなど、多くの有益なコメントをくださった査読者の皆さまに深く感謝いたします。

## 参考文献

[1] Martin Abadi, Luca Cardelli, P-L Curien, and J-J Lévy. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 31–46. ACM, 1989.

[2] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*, pages 18–29. ACM, 2016.

[3] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *International Workshop on Computer Science Logic*, pages 453–468. Springer, 1999.

[4] Andrew W Appel. *Compiling with continuations*. Cambridge University Press, 2007.

[5] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In *Asian Symposium on Programming Languages and Systems*, pages 239–254. Springer, 2007.

[6] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[7] Gilles Barthe, John Hatcliff, and Morten Heine B Sørensen. CPS translations and applications: the cube and beyond. *Higher-Order and Symbolic Computation*, 12(2):125–170, 1999.

[8] Gilles Barthe and Tarmo Uustalu. CPS translating inductive and coinductive types. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '02)*, pages 131–142. ACM, 2002.

[9] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[10] William Clinger, Daniel P Friedman, and Mitchell Wand. *A scheme for a higher-level semantic algebra*. Computer Science Department, Indiana University, 1983.

[11] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM, 1990.

[12] Zaynah Dargaye and Xavier Leroy. Mechanized verification of CPS transformations. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 211–225. Springer, 2007.

[13] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. 75(5):381–392, 1972.

[14] Peter Dybjer. Internal type theory. In *International Workshop on Types for Proofs and Programs*, pages 120–134. Springer, 1995.

[15] Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. *LISP and Symbolic Computation*, 6(3-4):361–379, 1993.

[16] Hugo Herbelin. On the degeneracy of $\sigma$-types in presence of computational classical logic. In *International Conference on Typed Lambda Calculi and Applications*, pages 209–220. Springer, 2005.

[17] Albert R Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi. In *Workshop on Logic of Programs*, pages 219–224. Springer, 1985.

[18] Gordon D. Plotkin. Call-by-name, call-by-value and the $\lambda$-calculus. *Theoretical computer science*, 1(2):125–159, 1975.

[19] Aaron Stump. *Verified Functional Programming in Agda*. Morgan & Claypool, 2016.

[20] Yuki Watanabe. Study on proof of type preservation in CPS transformation. Master's thesis, University of Tokyo, 2011.