AgdaによるPHOASを用いたCPS変換の正当性の証明

石尾 千晶, 山田 麗, 浅井 健一

お茶の水女子大学

ishio@pllab.is.ocha.ac.jp, yamada.urara@is.ocha.ac.jp, asai@is.ocha.ac.jp

概 要　継続とは、ある時点における残りの計算のことである。プログラム内で継続を扱う方法の一つに、プログラム全体を継続渡し形式 (Continuation-Passing Style; CPS) に変換するというものがある。本研究では、単純型付き λ 計算における Danvy/Filinski による one-pass の CPS 変換の正当性を定理証明支援系 Agda によって定式化し、その正当性を証明する。さらに、この証明が let 多相を含む体系でも成り立つよう拡張を行う。このとき、変数名を明示的に扱うのではなく、変数の束縛をメタ言語で行う PHOAS (Parameterized Higher-Order Abstract Syntax) を用いて証明することにより、手で書くのと同程度の複雑さで定式化することができた。

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。例えば、$1 + (2 * 3)$ という計算で、$2 * 3$ を実行しているときの継続は、「$2 * 3$ の計算結果を受け取ったら、そこに 1 を加える」ということになる。プログラムの中で継続を扱う方法に、プログラム全体を継続渡し形式 (Continuation-Passing Style; CPS) に変換するというものがある。CPS 変換をするとプログラム中の継続を管理できるため、コンパイラの中間言語に使われる [1] など数々の応用がある。

本稿は、selective CPS 変換 [13] の Agda [17] による定式化を試みる研究の一環で、単純型付き λ 計算に let 多相と限定継続のオペレータ `shift/reset` [8] を加えた体系 [2,8] での selective CPS 変換の証明 [3] の定式化を目指している。`shift/reset` は限定継続を取り扱うためのオペレータであり、`shift` が「その時点の継続を切り取ってくる命令」で、`reset` が「切り取られる継続の範囲を限定する命令」である。`shift/reset` には、非決定性プログラミング [8] や let insertion を用いた部分評価 [11] などいろいろな応用例がある。`shift/reset` を使ったプログラムは CPS 変換を行えば `shift/reset` をサポートしない言語でも実行可能だが、プログラム全体を CPS 変換すると効率が落ちる。selective CPS 変換では、`shift/reset` に必要な部分だけ CPS 変換することで `shift/reset` に関係ない部分の効率を落とすことなく実行できるようになる [3]。

`shift/reset` を導入するためには、現在のコンテキストの型をあらわす answer type についても各種の証明をしなければならない。さらに、selective CPS 変換は与えられた項の answer type が変化するかどうかによって数多くの場合分けをする必要があり、手で証明するのは次第に困難になってくる。そのような証明を Agda で行えるようにするべく、本稿ではその基礎となる let 多相の入った単純型付き λ 計算を対象にして、その CPS 変換の正当性の証明を Agda で行う。

これまで、各種のプログラミング言語理論の定式化を行うためには、変数束縛の問題を細部にわたって正確に管理しなければならなかった。しかし、PHOAS (Parameterized Higher-Order Abstract Syntax) [5] を使うと、変数をある程度、明示的に扱いつつも変数束縛をメタ言語で取り扱うことができるようになるため、変数束縛の管理にそれほど大きな労力を割くことなく定式化できる。特に、多相の入った体系でも PHOAS の利点を生かして証明できるという見通し [5] が得られており、selective CPS 変換の定式化への基盤として利用するには PHOAS が優れていると考えられる。本稿では、実際に CPS 変換の正当性の証明を行ってみた様子を報告し、let 多相の入っていない体系では、十分に簡潔で見通しの良い形で証明できることを示す。

一方、PHOAS は HOAS (Higher-Order Abstract Syntax) の一つなので、Agda で λ 式の中を分解できないなどの困難な点もある。実際、let 多相の入った体系の CPS 変換の証明を行う際には、自明に成り立つと思われる性質を PHOAS を使った定式化では示すことができず、仮定せざるを得なかった。その様子も本稿で報告する。

PHOAS はすでに Chlipala 自身によってコンパイラの正当性の表示的意味論に基づいた証明に使用されている。その中で System F の CPS 変換が扱われており [5]、その意味では本稿の結果は予想されるものである。しかし、Danvy/Filinski の one-pass の CPS 変換の正当性の証明 [9] をほぼそのまま Agda で定式化できており、PHOAS の適用範囲をみるケーススタディとしては一定の価値があると思われる。また、let 多相の定式化は System F とは違った形の定式化になっており、そこには若干の新規性があると考えている。let 多相の定式化には、仮定が残っているなど HOAS 特有の問題点は存在するが、それを除くと十分に簡潔な証明になっており、今後、限定継続命令への拡張などに道を開くものとなっている。

以下、2 節で単純型付き λ 計算を、3 節で Danvy/Filinski の one-pass の CPS 変換を Agda を用いて定式化した上で、4 節で CPS 変換の正当性を証明する。加えて、5 節で単純型付き λ 計算に let 多相を加えて定式化し、PHOAS 特有の問題はあるものの、仮定を立てることで証明を試みる。また、6 章で関連研究を、7 節でまとめと今後の課題を述べる。

本稿で示した証明の Agda のコードは `http://pllab.is.ocha.ac.jp/~asai/jpapers/ppl/18/` に置いている。

2 単純型付き λ 計算の定式化

この節では、対象言語を導入し、その PHOAS を使った Agda による定式化について述べる。

2.1 PHOAS

Agda で定式化するにあたって、PHOAS (Parameterized Higher-Order Abstract Syntax) を用いる。PHOAS とは、変数束縛のある言語の抽象構文木を表現するための方法の一つである。

対象言語の変数束縛は通常、名前を使って行われる。このような一階の抽象構文は First-Order Abstract Syntax (FOAS) と呼ばれる。FOAS を使うと、簡単に変数を扱える反面、変数の α 同値性を自分で定義する必要がある。これは、ほとんどの場合とても煩雑で、その解決を目指して PoplMark Challenge [4] が提示されるなど多くの解が求められてきた。PoplMark Challenge は、多相の λ 計算に部分型付けを組み合わせた体系 $F_{<:}$ において、変数束縛や複雑な帰納法の問題などを取り扱うことを目的として設けられている。

Higher-Order Abstract Syntax (HOAS) は、変数束縛の問題を回避するひとつの方法で、名前解決を対象言語ではなくメタ言語 (本稿では Agda) で行う。メタ言語の変数束縛をそのまま対象言語の定義に使用するため、定義が簡潔になり、代入などの操作も行いやすくなる。一方で、メタ言語の関数をそのまま使用するため、関数の中身に従った場合分けができなくなるなどの問題点もある。PHOAS はその問題点を、変数をパラメータ化することで一部、緩和するものである。

2.2 単純型付き λ 計算

本稿で対象とする言語は純粋な単純型付き λ 計算に定数 (自然数) を加えたものである。その型と構文を図 1 に示す。型は自然数型か関数型のどちらかである。これは Agda では以下のように定義できる。

```
data typ : Set where
  Nat : typ -- 自然数型
  _⇒_ : typ → typ → typ -- 関数型
```

$$
\begin{array}{lll}
\text{型} & \tau & ::= \quad \mathsf{Nat} \mid \tau \to \tau \\
\text{値} & v & ::= \quad c \mid x \mid \lambda x.\, e \\
\text{項} & e & ::= \quad v \mid e_1\, e_2
\end{array}
$$

図 1. 単純型付き λ 計算

値は変数・自然数・λ 抽象の 3 つからなり、項は値か関数適用である。本稿では、このように定義される構文のうち型が付くもののみを扱う。Agda では、依存型を使って「型 τ を持つ項」などを表現できる。これを使って、次のように値と項を Agda で定義する。値と項は相互再帰的に定義されるため、mutual というキーワードを使って定式化する。

```
mutual
  data value[_]_ (var : typ → Set) : typ → Set where
    Var : {τ₁ : typ} → var τ₁ → value[ var ] τ₁ -- 変数
    Num : (n : ℕ) → value[ var ] Nat -- 自然数
    Fun : {τ₁ τ₂ : typ} → -- ラムダ抽象
      (e₁ : var τ₂ → term[ var ] τ₁) → value[ var ] (τ₂ ⇒ τ₁)

  data term[_]_ (var : typ → Set) : typ → Set where
    Val : {τ₁ : typ} → value[ var ] τ₁ → term[ var ] τ₁ -- 値
    App : {τ₁ τ₂ : typ} → -- 関数適用
      (e₁ : term[ var ] (τ₂ ⇒ τ₁)) → (e₂ : term[ var ] τ₂) → term[ var ] τ₁
```

value[var]τ と term[var]τ は、それぞれ τ 型の値、項を示す。ここで var は、変数を表すパラメータで typ を引数に取る。ここは PHOAS 特有の部分である。var は型環境のことだと考えても良い。Var x は、x が $var\,\tau_1$ 型のとき（x が型環境で τ_1 型を持つとき）τ_1 型の変数を表す。Num は、自然数 n を受け取ると無条件で Nat 型の定数となる。Fun は、「τ_2 型の変数を受け取ったら τ_1 型の項を返す関数」を受け取ったら、$(\tau_2 \Rightarrow \tau_1)$ 型の λ 抽象となる。ここで Fun は、Agda 上の関数を引数に取ることに注意しよう。対象言語の変数の束縛をメタ言語 (Agda) の束縛で表現しており、ここが HOAS になっている部分である。例えば、$\lambda x.\, \lambda y.\, x$ は Fun ($\lambda x \to$ Val (Fun ($\lambda y \to$ Var x))) と表される。通常の HOAS では引数も項となるが、ここでは引数の型は変数 $var\,\tau_1$ となっている。値と項全体が var でパラメータ化されているため、変数も独立した構文（構成子）として扱うことができるようになるとともに、関数の引数部分（negative な部分）に値や項自身が現れることを避け、Agda でも構文を定義できるようになっている。これが PHOAS である。最後に、App は、$(\tau_2 \Rightarrow \tau_1)$ 型の項と τ_2 型の項を受け取ると、全体として τ_1 型の関数適用となる。

上記の値、項の定義は単純型付き λ 計算の型規則をそのまま埋め込んだものになっている。このように構文を定義すると、型の付く構文のみを議論の対象とすることができる。

2.3 代入規則

のちに正当性の証明に β 簡約を利用するため、2.2 節の体系での代入規則を relation で定義したものを図 2 に示す。$(\lambda y.\, e)[v] \mapsto e'$ と書くと、「項 e の中に出てくる y を値 v で置き換えると項 e' となる」と読み、Agda では Subst $e\, v\, e'$ と書く。項 e の中に出てくる y と書くと、e の中に y が自由に現れている必要があるが、PHOAS を使っていると自由に現れる変数を表現できない。そこで $(\lambda y.\, e)$ のように、代入する式が入る場所 y を λ 式で抽象化した式を使っている。$(\lambda y.\, e)[v] \mapsto e'$ の各場合については、以下に示す Agda のコードを参照しながら説明する。代入先の value や term は、2.2 節では相互再帰によって定義されているため、代入規則もそれに倣った構造となる。

SubstVal は、「代入する場所が τ 型であるような τ_1 型の値」と τ 型の値を受け取ったら、値を代入した結果として τ_1 型の値を返す規則である。sVar= は、変数が代入先の変数だった場合で、v を

$$(\lambda y.\, y)[v] \mapsto v \qquad (\lambda y.\, x)[v] \mapsto x \qquad (\lambda y.\, c)[v] \mapsto c \qquad \frac{\forall x((\lambda y.\, (e\, y)\, x)[v] \mapsto e'\, x)}{(\lambda y.\, \lambda x.\, e\, y)[v] \mapsto \lambda x.\, e'}$$

$$\frac{(\lambda y.\, e_1\, y)[v] \mapsto e_1' \qquad (\lambda y.\, e_2\, y)[v] \mapsto e_2'}{(\lambda y.\, (e_1\, y)\, (e_2\, y))[v] \mapsto e_1'\, e_2'}$$

図 2. 代入規則

```
mutual
  data SubstVal {var : typ → Set} : {τ τ₁ : typ} →
    (var τ → value[ var ] τ₁) → value[ var ] τ → value[ var ] τ₁ → Set where
  sVar= : {τ : typ} → {v : value[ var ] τ} → -- 変数に代入する
    SubstVal (λ x → Var x) v v
  sVar≠ : {τ τ₁ : typ} → {x : var τ₁} → {v : value[ var ] τ} → -- 変数に代入しない
    SubstVal (λ y → Var x) v (Var x)
  sNum  : {τ : typ} → {n : ℕ} → {v : value[ var ] τ} → -- 自然数には代入しない
    SubstVal (λ y → Num n) v (Num n)
  sFun  : {τ τ₁ τ₂ : typ} → -- ラムダ抽象に代入する
    {e₁ : var τ → var τ₂ → term[ var ] τ₁} → {v : value[ var ] τ} →
    {e₁' : var τ₂ → term[ var ] τ₁} →
    ((x : var τ₂) → Subst (λ y → (e₁ y) x) v (e₁' x)) →
    SubstVal (λ y → Fun (e₁ y)) v (Fun e₁')

  data Subst {var : typ → Set} : {τ τ₁ : typ} →
    (var τ → term[ var ] τ₁) → value[ var ] τ → term[ var ] τ₁ → Set where
  sVal : {τ τ₁ : typ} → -- 値に代入する
    {v₁ : var τ → value[ var ] τ₁} → {v : value[ var ] τ} → {v₁' : value[ var ] τ₁} →
    SubstVal v₁ v v₁' →
    Subst (λ y → Val (v₁ y)) v (Val v₁')
  sApp : {τ τ₁ τ₂ : typ} → -- 関数適用に代入する
    {e₁ : var τ → term[ var ] (τ₂ ⇒ τ₁)} → {e₂ : var τ → term[ var ] τ₂} →
    {v : value[ var ] τ} → {e₁' : term[ var ] (τ₂ ⇒ τ₁)} → {e₂' : term[ var ] τ₂} →
    Subst e₁ v e₁' → Subst e₂ v e₂' →
    Subst (λ y → App (e₁ y) (e₂ y)) v (App e₁' e₂')
```

x に代入して値 v を返す。sVar≠ は、変数が代入先の変数とは異なった場合で、値 v は代入されず変数 x を返す。sNum は、自然数には代入先の変数は現れないので元の自然数がそのまま返ることを示す。sFun は、少し面倒な形をしているが、以下のように理解できる。まず、λ 抽象の本体部分 $e\, y$ はもともと PHOAS のため引数 x を受け取るような関数で表現されている。そこに、さらに代入する場所を示すために y が導入されるため、e は結果として y と x を受け取るような関数となる。その y 部分に v を代入するためには、任意の x について $(e\, y)\, x$ の y 部分に v を代入した結果を得れば良い。ここで、任意の x を選ぶ部分はメタ言語で行われていることに注意しよう。ここでも HOAS の考え方が使われている。

次に Subst を見る。sVal では、値 v_1 の中に現れる y に値 v を代入したら値 v_1' になることを SubstVal を用いて定義している。sApp では、項 e_1, e_2 の中に現れる y に値 v を代入したらそれぞれ項 e_1', e_2' になるとき、関数適用 $e_1\, e_2$ は $e_1'\, e_2'$ になることを示している。

上記の代入規則は型付きで定義されているため、代入規則を Agda で定義できた時点で次の代入補題 [18] が示されたことになる。

補題 1 (代入の下での型の保存) $\Gamma, y : \tau' \vdash e : \tau$ かつ $\Gamma \vdash v : \tau'$ ならば $\Gamma \vdash (\lambda y.\, e)[v] : \tau$ である。

本研究では代入規則を relation として定義しているが、これ以外の方法としては Agda の関数を用いて定義する [5,6] ことも考えられる。関数による定義は、直接、関数の結果として代入結果を得られるが、その際 var を具体化する必要がある。すると、small-step の意味論では、簡約をするた

びに var の具体化と一般化を繰り返さなければならず、煩雑になる。そのように定式化しても証明は可能かもしれないが、本論文では relation を使って代入規則を定義する方法をとった。Chlipala 自身も System F の定式化の際には型変数への代入を relation で定義している [5]。

2.4 簡約規則

代入規則に続いて β 簡約規則を定める。簡約は call-by-value で行うものとする。call-by-value とは、関数適用の際に引数を先に評価してから簡約をすることである。

まず β 簡約規則のための評価規則を図 3 に示すフレームと評価文脈を用いて定義する。[ ] はこれから評価を行う場所を意味する。フレームと、フレームの [ ] に項を入れる frame-plug は Agda で次のように書ける。

```
data frame[_,_] (var : typ → Set) : typ → typ → Set where
  App₁ : {τ₁ τ₂ : typ} → (e₂ : term[ var ] τ₂) → frame[ var , τ₂ ⇒ τ₁ ] τ₁
  App₂ : {τ₁ τ₂ : typ} → (v₁ : value[ var ] (τ₂ ⇒ τ₁)) → frame[ var , τ₂ ] τ₁

frame-plug : {var : typ → Set} → {τ₁ τ₂ : typ} →
    frame[ var , τ₂ ] τ₁ → term[ var ] τ₂ → term[ var ] τ₁
frame-plug (App₁ e₂) e₁ = App e₁ e₂
frame-plug (App₂ v₁) e₂ = App (Val v₁) e₂
```

frame$[var, \tau_2]\,\tau_1$ は「全体として τ_1 型になるような項で、中の [ ] が τ_2 型をもつフレーム」を意味する。App$_1$ は関数部分を実行中であることを示すフレームである。App $e_1\,e_2$ という関数適用があり e_1 が項のときは、e_1 が先に評価される。App$_2$ は引数部分を実行中であることを示すフレームである。App $v_1\,e_2$ という関数適用で v_1 が値、e_2 が項のとき、v_1 は評価済みなので e_2 が評価される。

次に、図 4 のような β 簡約規則を定義する。簡約規則には、1 ステップ簡約する Reduce と、0 ステップ以上簡約する Reduce*の 2 つの規則がある。

```
data Reduce {var : typ → Set} : {τ₁ : typ} → term[ var ] τ₁ → term[ var ] τ₁ → Set where
  RBeta  : {τ₁ τ₂ : typ} → -- β 簡約
    {e : (x : var τ₂) → term[ var ] τ₁} → {v₂ : value[ var ] τ₂} → {e' : term[ var ] τ₁} →
    Subst e v₂ e' → Reduce (App (Val (Fun e)) (Val v₂)) e'
  RFrame : {τ₁ τ₂ : typ} → -- CBV による部分式の評価
    {e e' : term[ var ] τ₁} → (f : frame[ var , τ₁ ] τ₂) →
    Reduce e e' → Reduce (frame-plug f e) (frame-plug f e')

data Reduce* {var : typ → Set} : {τ₁ : typ} → term[ var ] τ₁ → term[ var ] τ₁ → Set where
  R*Id  : {τ₁ : typ} → (e : term[ var ] τ₁) → Reduce* e e -- 0 ステップの簡約
  R*Trans : {τ₁ : typ} → (e₁ e₂ e₃ : term[ var ] τ₁) → -- 1 ステップ以上の簡約
    Reduce e₁ e₂ → Reduce* e₂ e₃ → Reduce* e₁ e₃
```

RBeta は、関数適用を実行する。ここで、項 e は x を受け取って τ_1 型の項を返すような関数抽象である。e の中に出てくる x に値 v_2 を代入した結果が e' になるとき、(App (Val (Fun e))(Val v_2)) を実行すると e' となる。RFrame はフレームを用いた評価規則に基づいて項を評価する。ここで frame-plug $f\,e$ は項 e にフレーム f をつけることを意味していたことに注意すると、項 e を簡約した結果が e' になるとき、$f[e]$ は $f[e']$ に簡約される。

次に Reduce*を見てみる。Reduce*は Reduce の反射推移閉包である。R*Id は、受け取った項 e は 0 ステップで自分自身となることを意味する。R*Trans は、1 ステップ以上、簡約を進める規則である。項 e_1 を 1 ステップ簡約した結果が項 e_2 となり、項 e_2 を 0 ステップ以上簡約した結果が e_3 になるとき、e_1 を 1 ステップ以上簡約した結果が e_3 になることを意味する。

ここでも、簡約規則がすべて型付きで定義されているため、簡約規則を Agda で書けたことで、項を簡約した結果は型が保存されていること、つまり subject reduction [16] が証明されたことになる。

$$
\begin{array}{lrcl}
\text{フレーム} & F & = & [\,]\,e \mid v\,[\,] \\
\text{評価文脈} & E & = & [\,] \mid F\,[E]
\end{array}
$$

図 3. 評価文脈

$$
\frac{(\lambda x.\, e)[v] \mapsto e'}{(\lambda x.\, e)\, v \to e'} \qquad \frac{e \to e'}{F[e] \to F[e']}
$$

図 4. 簡約規則

3 CPS 変換の定式化

本稿では、Danvy/Filinski による one-pass の CPS 変換 [9] を扱う。ここでは call-by-value かつ left-to-right の CPS 変換を扱う。型の変換を図 5 に、項の変換を図 6 に示す。本稿では、answer type は Nat に固定する。

$$
\begin{array}{rcl}
\mathsf{Nat}^* & = & \mathsf{Nat} \\
(\tau_1 \Rightarrow \tau_2)^* & = & {\tau_1}^* \Rightarrow ({\tau_2}^* \Rightarrow \mathsf{Nat}) \Rightarrow \mathsf{Nat}
\end{array}
$$

図 5. 型の CPS 変換

$$
\begin{array}{rcl}
[\![c]\!]_v & = & c \\
[\![x]\!]_v & = & x \\
[\![\lambda x.\, e]\!]_v & = & \underline{\lambda} x.\, \underline{\lambda} k.\, [\![e]\!]'\, \overline{@}\, k \\
\\
[\![v]\!] & = & \overline{\lambda} \kappa.\, \kappa\, \overline{@}\, [\![v]\!]_v \\
[\![e_1\, @\, e_2]\!] & = & \overline{\lambda} \kappa.\, [\![e_1]\!]\, \overline{@}\, (\overline{\lambda} m.\, [\![e_2]\!]\, \overline{@}\, (\overline{\lambda} n.\, (m\, \underline{@}\, n)\, \underline{@}\, (\underline{\lambda} a.\, \kappa\, \overline{@}\, a))) \\
\\
[\![v]\!]' & = & \overline{\lambda} k.\, k\, \underline{@}\, [\![v]\!]_v \\
[\![e_1\, @\, e_2]\!]' & = & \overline{\lambda} k.\, [\![e_1]\!]\, \overline{@}\, (\overline{\lambda} m.\, [\![e_2]\!]\, \overline{@}\, (\overline{\lambda} n.\, (m\, \underline{@}\, n)\, \underline{@}\, k))
\end{array}
$$

図 6. 単純型付き λ 計算の CPS 変換

CPS 変換は、値用の $[\![v]\!]_v$ と項用の $[\![e]\!]$ と $[\![e]\!]'$ の 3 種類からなる。CPS 変換の結果は、アンダーラインのついた式とオーバーラインのついた式を使った 2 レベルの λ 計算で書かれている。アンダーラインのついた式は dynamic な式と呼ばれ、出力の構文を表す。Agda では Fun や App などの構成子で表現される。一方、オーバーラインのついた式は static な式と呼ばれ、CPS 変換時にこれらの命令は実行されることを示す。これらは Agda の（実行可能な）関数や関数呼び出しとなる。static な命令を CPS 変換時に実行してしまうため、いわゆる administrative β-redex を生じることなく、コンパクトな CPS 変換結果を生成することができる [9]。

$[\![e]\!]\,\kappa$ は、項 e を受け取ると、それを static な初期継続 κ のもとで CPS 変換した結果を返すことを意味する。このとき κ はメタ言語で書かれる関数であり、変換時に実行される。一方、$[\![e]\!]'\,k$ は、項 e を受け取ると、それを dynamic な初期継続 k のもとで CPS 変換した結果を返すことを意味する。このように 2 種類の変換を用意しているのは、CPS 変換時に継続がわかっている場合とわかっていない場合があるためである。関数適用の規則では継続は具体的な形で与えられているが、λ 抽象の規則では継続の形は不明である。このように継続の形がわかっているかどうかで 2 種類の変換を用意することで、administrative η-redex を生じることなく CPS 変換結果を得ることができるようになる [9]。

上の規則を Agda で実装すると以下のようになる。規則をほぼそのまま Agda に直せていることがわかる。cpsEta と $\mathsf{cpsEta'}$ の継続は、前者は static なため Agda の関数型を持ち、後者は dynamic なため（関数型を持つ）値となっている。PHOAS 特有のこととして、λ 抽象の場合は少し説明が必要である。PHOAS を使っているため、Fun の引数である e_1 は static な関数である。また、CPS

```
-- 型レベルの CPS 変換
cpsT : typ → typ
cpsT Nat = Nat
cpsT (τ₂ ⇒ τ₁) = cpsT τ₂ ⇒ (cpsT τ₁ ⇒ Nat) ⇒ Nat

mutual
  -- 値に対する CPS 変換
  cpsEtaV : {var : typ → Set} → {τ : typ} → value[ var ∘ cpsT ] τ → value[ var ] cpsT τ
  cpsEtaV (Var x) = Var x
  cpsEtaV (Num n) = Num n
  cpsEtaV (Fun e₁) = Fun (λ x → Val (Fun (λ k → cpsEta′ (e₁ x) (Var k))))

  -- 項に対する CPS 変換（static な継続）
  cpsEta : {var : typ → Set} → {τ : typ} →
    term[ var ∘ cpsT ] τ → (value[ var ] (cpsT τ) → term[ var ] Nat) → term[ var ] Nat
  cpsEta (Val v) κ = κ (cpsEtaV v)
  cpsEta (App e₁ e₂) κ =
    cpsEta e₁ (λ v₁ → cpsEta e₂ (λ v₂ →
      App (App (Val v₁) (Val v₂)) (Val (Fun (λ v → κ (Var v))))))

  -- 項に対する CPS 変換の補助規則（dynamic な継続）
  cpsEta′ : {var : typ → Set} → {τ : typ} →
    term[ var ∘ cpsT ] τ → value[ var ] (cpsT τ ⇒ Nat) → term[ var ] Nat
  cpsEta′ (Val v) k = App (Val k) (Val (cpsEtaV v))
  cpsEta′ (App e₁ e₂) k =
    cpsEta e₁ (λ v₁ → cpsEta e₂ (λ v₂ → App (App (Val v₁) (Val v₂)) (Val k)))
```

変換結果の Fun も引数として static な関数を受け取る必要がある。CPS 変換結果の関数の引数 x を e_1 に渡すことで e_1 の本体を得て、それを CPS 変換にかけている。この変換を通じて、変数の束縛がメタ言語の変数の束縛を通じてうまく表現されていることがわかる。

上記の Agda による CPS 変換は、変換される項の型情報も含めて定義されているため、実装ができた時点で、次の定理が示されたことになる。

定理 1 (CPS 変換の型保存) $\Gamma \vdash v : \tau$ ならば $\Gamma^* \vdash [\![v]\!]_v : \tau^*$ が成り立つ。$\Gamma \vdash e : \tau$ ならば $\Gamma^* \vdash [\![e]\!] : (\tau^* \to \mathsf{Nat}) \to \mathsf{Nat}$ かつ $\Gamma^* \vdash [\![e]\!]' : (\tau^* \Rightarrow \mathsf{Nat}) \to \mathsf{Nat}$ が成り立つ。

CPS 変換の例を考える。$[\![3]\!]$ は $\overline{\lambda}\kappa.\, \kappa\, \overline{@}\, 3$ になるが、これは $\mathsf{cpsEta}\,(\mathsf{Val}\,(\mathsf{Num}\,3))$ を実行すると $\lambda\kappa \to \kappa\,(\mathsf{Num}\,3)$ になるのと等しい。また、$[\![(\underline{\lambda}x.\, x)\, \underline{@}\, 3]\!]$ は

$$\overline{\lambda}\kappa.\, ((\underline{\lambda}x.\, \underline{\lambda}k.\, k\, \underline{@}\, x)\, \underline{@}\, 3)\, \underline{@}\, (\underline{\lambda}v.\, \kappa\, \overline{@}\, v)$$

になる。これは、$\mathsf{cpsEta}\,(\mathsf{App}\,(\mathsf{Val}\,(\mathsf{Fun}\,(\lambda x \to (\mathsf{Val}\,(\mathsf{Var}\,x)))))\,(\mathsf{Val}\,(\mathsf{Num}\,3)))$ が

$$\lambda\kappa \to \mathsf{App}\,(\mathsf{App}\,(\mathsf{Val}\,(\mathsf{Fun}\,(\lambda x \to \mathsf{Val}\,(\mathsf{Fun}\,(\lambda k \to \mathsf{App}\,(\mathsf{Val}\,(\mathsf{Var}\,k))\,(\mathsf{Val}\,(\mathsf{Var}\,x))))))))\,(\mathsf{Val}\,(\mathsf{Num}\,3)))\,(\mathsf{Val}\,(\mathsf{Fun}\,(\lambda v \to \kappa\,(\mathsf{Var}\,v))))$$

となるのと同じである。

4 単純型付き λ 計算の CPS 変換の正当性の証明

この節では、3 節で示した CPS 変換が正しいこと、つまり CPS 変換をしても、変換前の項と同等の簡約を行うことを示す。最終的に示される定理は以下のようになる。

定理 2 (CPS 変換の正当性) 任意の項 e, e' について $e \to e'$ が成り立つなら、任意の schematic な継続 κ について $[\![e]\!]\, \overline{@}\, \kappa \to^* [\![e']\!]\, \overline{@}\, \kappa$ が成り立つ。

ここで、$e \to e'$ は入力言語での β 簡約、$[\![e]\!]\,\overline{@}\,\kappa \to^* [\![e']\!]\,\overline{@}\,\kappa$ は出力言語（アンダーラインのついた項）での β 簡約 である。継続が schematic であることの定義は、4.1 節で述べる。上の定理は、Agda では以下のように書ける。

```
correctEta : {var : typ → Set} → {τ : typ} → {e e′ : term[ var ∘ cpsT ] τ} →
  (κ : value[ var ] cpsT τ → term[ var ] Nat) →
  Reduce e e′ → schematic κ → Reduce* (cpsEta e κ) (cpsEta e′ κ)
```

4.1 schematic

定理 2 は任意の継続 κ で成り立つわけではない。例えば、e が $(\lambda x.\, x)\,@\,3$、e' が 3 だったとしよう。static な継続 κ のもとでのそれぞれの CPS 変換は、3 節で述べたように

$$\begin{array}{rclcl} [\![e]\!]\kappa & = & [\![(\lambda x.\, x)\,@\,3]\!]\kappa & = & ((\underline{\lambda} x.\, \underline{\lambda} k.\, k\,\underline{@}\,x)\,\underline{@}\,3)\,\underline{@}\,(\underline{\lambda} v.\, \kappa\,\overline{@}\,v) \\ [\![e']\!]\kappa & = & [\![3]\!]\kappa & = & \kappa\,\overline{@}\,3 \end{array}$$

となる。ここで $[\![e]\!]\kappa \to^* [\![e']\!]\kappa$ となるかを調べてみると、$[\![e]\!]\kappa$ に対して 3 回、dynamic な β 簡約 をすると $[\![e']\!]\kappa$ が得られるので、任意の κ に対して定理が成り立っているように見える。しかし、ここで注意しなくてはいけないのは上の CPS 変換結果に現れる $\kappa\,\overline{@}\,v$ と $\kappa\,\overline{@}\,3$ は static な関数適用で、CPS 変換時に簡約されるということである。ここで、例えば κ が「渡された引数が変数だったら 1 を返し、それ以外だったら 2 を返す」ような継続 κ_0 だったとすると、上の結果は以下のようになってしまう。

$$\begin{array}{rclcl} [\![e]\!]\kappa_0 & = & [\![(\lambda x.\, x)\,@\,3]\!]\kappa_0 & = & ((\underline{\lambda} x.\, \underline{\lambda} k.\, k\,\underline{@}\,x)\,\underline{@}\,3)\,\underline{@}\,(\underline{\lambda} v.\, 1) \\ [\![e']\!]\kappa_0 & = & [\![3]\!]\kappa_0 & = & 2 \end{array}$$

これでは明らかに $[\![e]\!]\kappa_0$ を簡約しても $[\![e']\!]\kappa_0$ には至らない。

上の例で定理が成り立たなかったのは κ が引数の変数を単なるデータとしてとらえており、値の代入などが起こりうる構文木としての性質を考慮していなかったためである。そのような「引数の syntactic な構造に独立な継続（構造を変更しないような継続）」のことを schematic な継続と呼ぶ [9]。この性質は以下のように書ける。

$$(\lambda y.\, \kappa\,\overline{@}\,y)[v] \mapsto \kappa\,\overline{@}\,v$$

つまり、κ に y を渡した結果の項の y の部分に v を入れたものは、最初から κ に v を渡したものと等しくなるということである。

これを Agda で表現すると以下のようになる。

```
schematic : {var : typ → Set} → {τ : typ} →
  (κ : value[ var ] cpsT τ → term[ var ] Nat) → Set
schematic {var} {τ} κ = (v : value[ var ] cpsT τ) → Subst (λ y → κ (Var y)) v (κ v)
```

4.2 正当性の証明

定理 2 の証明の概略を示す。定理 2 は、Reduce $e\,e'$ の構造によって場合分けをして証明する。最初の場合分けは帰納法の base case に相当し、e が $(\underline{\lambda} x.\, e_1)\,\underline{@}\,v$ の形のとき β 簡約をする。残りのケースはフレームの中が簡約される場合で、フレームの形によってさらに場合分けし、どちらも再帰する形で証明する。

各場合の証明は、手で書いた証明に近い形で書けるよう、equational reasoning に似せて証明を書いた。結論の Reduce* 関係を示すときは、全体を begin と ■ で囲み、中に簡約したい式を順に書き、各式の間に式変形の根拠を書く。式変形の根拠は 3 種類あり、$\equiv\langle\,\rangle$ は前後の式が等しいことを、$\to\langle\,\rangle$ は 1 ステップで簡約できることを、$\to^*\langle\,\rangle$ は 0 ステップ以上で簡約できることを意味する。いずれも $\langle$ と $\rangle$ の間に証明項を書く。

base case の証明は以下のようになる。

$$\begin{array}{lll} [\![(\underline{\lambda} x.\, e_1)\,\underline{@}\,v]\!]\,\kappa & = & ((\underline{\lambda} x.\, \underline{\lambda} k'.\, [\![e\,x]\!]'\,k')\,\underline{@}\,[\![v_2]\!]_v)\,\underline{@}\,(\underline{\lambda} v.\, \kappa\,\overline{@}\,v) \\ & \to & (\underline{\lambda} k'.\, [\![e']\!]'\,k')\,\underline{@}\,(\underline{\lambda} v.\, \kappa\,\overline{@}\,v) \\ & \to & [\![e']\!]'\,(\underline{\lambda} v.\, \kappa\,\overline{@}\,v) \\ & \to^* & [\![e']\!]\,\kappa \end{array}$$

上の数式をそのまま Agda で定式化することができている。

```
correctEta {e′ = e′} κ (RBeta {e = e} {v₂} sub) sch-k = -- base case
  begin
    cpsEta (App (Val (Fun e)) (Val v₂)) κ
  ≡⟨ refl ⟩
    App
      (App (Val (Fun (λ x → Val (Fun (λ k′ → cpsEta′ (e x) (Var k′)))))) (Val (cpsEtaV v₂)))
      (Val (Fun (λ v → κ (Var v))))
  ⟶⟨ RFrame (App₁ (Val (Fun (λ v → κ (Var v))))) (RBeta (sVal (sFun (λ k′ → eSubst′ k′ sub)))) ⟩ -- 1 ステップ簡約
    frame-plug (App₁ (Val (Fun (λ v → κ (Var v))))) (Val (Fun (λ k′ → cpsEta′ e′ (Var k′))))
  ≡⟨ refl ⟩
    App (Val (Fun (λ k′ → cpsEta′ e′ (Var k′)))) (Val (Fun (λ v → κ (Var v))))
  ⟶⟨ RBeta (kSubst′ e′) ⟩ -- 1 ステップ簡約
    cpsEta′ e′ (Fun (λ v → κ (Var v)))
  ⟶*⟨ cpsEtaEta′ e′ κ sch-k ⟩ -- 0 または 1 ステップ簡約
    cpsEta e′ κ
  ■
```

base case に注目すると、$\to\langle\,\rangle$ が 2 回使われている。このうち最初に出てくる $\to\langle\,\rangle$ では、引数 x として (Val (cpsEtaV v_2)) を受け取ってきて代入し、$(e\,x)$ を最終的には e' に簡約している。しかし x は CPS 変換規則の cpsEta′ の引数の中に現れるため 2.3 節で定めた代入規則では代入ができず簡約に進めない。これを解決するため eSubst′ という補題を使っている。また、次に出てくる $\to\langle\,\rangle$ では、引数 k' として (Val (Fun ($\lambda v \to k$ (Var v)))) を受け取り代入しようするが、前の例と同様 k' が cpsEta′ の引数の中に現れるため代入ができない。そのため、kSubst′ という補題を示して利用している。以下に示した eSubst′ と kSubst′ は、いずれも CPS 変換規則の cpsEta′ にかかわる補題だが、3 節で示したように CPS 変換規則は相互再帰的に 3 種類あるので、補題もそれにしたがってそれぞれ 3 つずつ定める必要がある。

補題 2 (CPS 変換と代入演算の可換性 1) 任意の項 e, e'、値 v、dynamic な継続 k について $(\lambda y.\, e\,y)[v] \mapsto e'$ が成り立つとき、$(\lambda y.\, [\![e\,y]\!]'\,k)[[\![v]\!]_v] \mapsto [\![e']\!]'\,k$ が成り立つ。

```
eSubst′ : {var : typ → Set} → {τ₁ τ₂ : typ} →
  {e : var (cpsT τ₂) → term[ var ∘ cpsT ] τ₁} →
  {e′ : term[ var ∘ cpsT ] τ₁} → {v : value[ var ∘ cpsT ] τ₂} →
  (k : var (cpsT τ₁ ⇒ Nat)) →
  Subst e v e′ →
  Subst (λ y → cpsEta′ {var} (e y) (Var k)) (cpsEtaV v) (cpsEta′ e′ (Var k))
```

補題 3 (CPS 変換と代入演算の可換性 2) 任意の項 e、値 v、dynamic な継続 k について $(\lambda k.\, [\![e\,y]\!]'\,k)[v] \mapsto [\![e\,y]\!]'\,v$ が成り立つ。

```
kSubst′ : {var : typ → Set} → {τ : typ} →
  (e : term[ var ∘ cpsT ] τ) → {v : value[ var ] (cpsT τ ⇒ Nat)} →
  Subst (λ k₁ → cpsEta′ e (Var k₁)) v (cpsEta′ e v)
```

base case の最後に現れる $\to^*\langle\,\rangle$ では、e' の構造によって簡約のステップ数が異なるため以下に示す cpsEtaEta′ という補題で別途証明をしている。e' が value のときは 1 ステップの簡約が必要で、

そうでないとき簡約は必要ない。このことから、一番初めに与えられた式 (cpsEta $e\,k$) を 2 ステップもしくは 3 ステップ簡約することで (cpsEta $e'\,k$) に到達することがわかる。

補題 4 (0 または 1 ステップの簡約) 任意の項 e、schematic な継続 κ について $[\![e]\!]'\,(\underline{\lambda} v.\,\kappa\,v) \to^* [\![e]\!]\,\kappa$ が成り立つ。

```
cpsEtaEta′ : {var : typ → Set} → {τ : typ} → -- 0 または 1 ステップの簡約
  (e : term[ var ∘ cpsT ] τ) →
  (κ : value[ var ] cpsT τ → term[ var ] Nat) → schematic κ →
  Reduce* (cpsEta′ e (Fun (λ v → κ (Var v)))) (cpsEta e κ)
```

次に、フレームの中が簡約される場合は以下のようになり、これも Agda でそのままコードにすることができた。

- inductive case 1

$$\begin{aligned}
[\![[\,e\,]\,e_2]\!]\,\kappa &= [\![e]\!]\,(\overline{\lambda} v_1.\,[\![e_2]\!]\,(\overline{\lambda} v_2.\,(v_1\,\underline{@}\,v_2)\,\underline{@}\,(\underline{\lambda} v.\,\kappa\,\overline{@}\,v))) \\
&\to [\![e']\!]\,(\overline{\lambda} v_1.\,[\![e_2]\!]\,(\overline{\lambda} v_2.\,(v_1\,\underline{@}\,v_2)\,\underline{@}\,(\underline{\lambda} v.\,\kappa\,\overline{@}\,v))) \quad \text{(IH)} \\
&= [\![[\,e'\,]\,e_2]\!]\,\kappa
\end{aligned}$$

- inductive case 2

$$\begin{aligned}
[\![v_1\,[\,e\,]]\!]\,\kappa &= [\![e]\!]\,(\overline{\lambda} v_2.\,([\![v_1]\!]_v\,\underline{@}\,v_2)\,\underline{@}\,(\underline{\lambda} v.\,\kappa\,\overline{@}\,v)) \\
&\to [\![e']\!]\,(\overline{\lambda} v_2.\,([\![v_1]\!]_v\,\underline{@}\,v_2)\,\underline{@}\,(\underline{\lambda} v.\,\kappa\,\overline{@}\,v)) \quad \text{(IH)} \\
&= [\![v_1\,[\,e'\,]]\!]\,\kappa
\end{aligned}$$

5 let 多相による拡張

この節では、今までの節で定式化した単純型付き λ 計算の定式化を、let 多相によって拡張した定式化について述べる。

5.1 多相型

let 多相の定式化をするためにはまず、型を多相に変更する必要がある。多相の型は、計算が行われるまでは「任意の型 α」を持ち、具体的な項において型付けを行う際に単相の型へ具体化される。

2 節で定めた型 typ に型変数を加えたものを単相の型とし、さらに型スキームを定義する。型スキームは、単相の型か多相の型からなる。Agda による型と型スキームの定義を以下に示す。

```
data typ[_] (tvar : Set) : Set where
  TVar : tvar → typ[ tvar ] -- 型変数
  Nat : typ[ tvar ] -- 自然数型
  _⇒_ : typ[ tvar ] → typ[ tvar ] → typ[ tvar ] -- 関数型

data ts[_] (tvar : Set) : Set where
  Typ : typ[ tvar ] → ts[ tvar ] -- 単相型
  Forall : (ts : tvar → ts[ tvar ]) → ts[ tvar ] -- 多相型
```

2 節の型定義と異なり、typ は $tvar$ でパラメータ化されている。この $tvar$ は、型変数を表しており、型変数 TVar は $tvar$ を受け取るように定義される。型スキーム ts も、$tvar$ でパラメータ化されている。多相の型は Forall で定義されていて、型変数を受け取ったら型スキームを返す、という Agda 上の関数で定義され、メタ言語の束縛で型変数の束縛を表現している。

$$
\begin{array}{rrcl}
\text{型} & \tau & ::= & \alpha \mid \mathsf{Nat} \mid \tau \to \tau \\
\text{型スキーム} & \sigma & ::= & \tau \mid \forall \alpha.\sigma \\
\text{値} & v & ::= & c \mid x \mid \lambda x.\, e \\
\text{項} & e & ::= & v \mid e_1\, e_2 \mid \mathsf{let}\ x = v\ \mathsf{in}\ e\, x
\end{array}
$$

図 7. let 多相を加えた単純型付き λ 計算

$$
\frac{\Gamma \vdash v_1 : \tau_1 \qquad \Gamma, x : \mathit{Gen}(\tau_1, \Gamma) \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = v_1\ \mathsf{in}\ e_2 : \tau_2}
\qquad
\frac{\forall \tau_1.(\sigma_1 > \tau_1 \to \Gamma \vdash v_1 : \tau_1) \qquad \Gamma, x : \sigma_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = v_1\ \mathsf{in}\ e_2\, x : \tau_2}
$$

図 8. Let 項の型付け規則

5.2 let 多相

多相の型を使って let 多相を定式化する。let 多相を加えた単純型付き λ 計算の型と型スキーム、そして構文を図 7 に示す。図 7 において、Let 項の定義は let $x = v_1$ in $e_2\, x$ のように e_2 が x を受け取るような形になっているが、これは e_2 の中に変数 x が表れていることを示すものである。本稿では、副作用は扱っていないが、将来的に限定継続命令を導入することを視野に入れて、let 文は value restriction を採用している。

以下に、値と項の Agda による定義を示す。

```
mutual
  data value[_]_ {tvar : Set} (var : ts[ tvar ] → Set) : typ[ tvar ] → Set where
    Var : {σ₁ : ts[ tvar ]} → (x : var σ₁) → {τ₁ : typ[ tvar ]} → -- 変数
      (p : inst σ₁ τ₁) → value[ var ] τ₁
    Num : (n : ℕ) → value[ var ] Nat -- 自然数
    Fun : {τ₁ τ₂ : typ[ tvar ]} → (e₁ : var (Typ τ₂) → term[ var ] τ₁) → -- ラムダ抽象
      value[ var ] (τ₂ ⇒ τ₁)

  data term[_]_ {tvar : Set} (var : ts[ tvar ] → Set) : typ[ tvar ] → Set where
    Val : {τ₁ : typ[ tvar ]} → value[ var ] τ₁ → term[ var ] τ₁ -- 値
    App : {τ₁ τ₂ : typ[ tvar ]} → -- 関数適用
      (e₁ : term[ var ] (τ₂ ⇒ τ₁)) → (e₂ : term[ var ] τ₂) → term[ var ] τ₁
    Let : (σ₁ : ts[ tvar ]) → {τ₂ : typ[ tvar ]} → -- 多相の let
      (v₁ : {τ₁ : typ[ tvar ]} → inst σ₁ τ₁ → value[ var ] τ₁) →
      (e₂ : var σ₁ → term[ var ] τ₂) → term[ var ] τ₂
```

2 節における値と項の定義では、var は型を受け取る関数であったが、多相の型を扱うため var は型スキームを受け取るようになっている。Var と Let の定義以外は、2 節の定義を型スキームを扱うように書き換えただけである。Var と Let において、多相の型を扱うための新しい関係 inst というものが表れている。inst $\sigma\ \tau$ は、多相の型 σ が単相の型 τ に具体化することができることを表す関係である。この関係を定義するため、型の代入も定義しているが、項の代入の定義とほとんど同じ方法で行なっているため、説明は省略する。これを踏まえて定義を見ていこう。Var は $var\ \sigma_1$ 型の変数 x を受け取ったら、σ_1 型が τ_1 型に具体化できるとき τ_1 型の変数を表す。Let では、まず型 σ_1 を受け取る。これは、Let によって定義される変数の多相の型を表している。次に v_1 を受け取るが、v_1 は多相の型 σ_1 を持つ値である。v_1 が確かに多相の型を持っていることを保証するため、v_1 は σ_1 と inst 関係にある任意の τ_1 型を持つようにしている。続いて e_2 を受け取る。e_2 は $var\ \sigma_1$ 型の変数を受け取ったら τ_2 型の項を返す関数である。それらを受け取ったら Let の項を表すという定義である。この定義は let $x = v_1$ in $e_2\, x$ を意味している。e_2 の項を Agda 上の関数として扱うことで、e_2 の中の変数 x の束縛を表現することができているとわかる。

本稿の定義における Let 項の型付け規則は、一般的な型付け規則とは少々異なる形をしている。図 8 は、左が一般的な Let 項の型付け規則で、右が本稿における Let 項の型付け規則である。ここ

で、$Gen(\tau_1, \Gamma)$ は、τ_1 に自由に表れていて Γ に含まれない型変数を束縛して多相の型として扱う関数を、$\sigma_1 > \tau_1$ は inst σ_1 τ_1 を表す。一般的な Let 項の型付け規則では、関数 Gen というものを定義して変数の束縛を関数で行うが、本稿では HOAS を採用しており、全ての変数の束縛はメタ言語のレベルで行われる。そこで、本稿では多相の型から始めて、それを具体化したどの型も持つ、という逆のアプローチで型付け規則を定義した。

5.3 CPS 変換後の項

let 多相の型付け規則を前章のように行なったことで、CPS 変換の前後で項の定義を別に行う必要が出てきた。その理由として、まず Let 項の CPS 変換の定義を見ていく。

$$[\![\mathsf{let}\ x = v\ \mathsf{in}\ e\,x]\!] \quad = \quad \overline{\lambda}\kappa.\,\underline{\mathsf{let}}\ x = [\![v]\!]_v\ \ \underline{\mathsf{in}}\ \ [\![e\,x]\!]\,\overline{@}\,\kappa$$

$$[\![\mathsf{let}\ x = v\ \mathsf{in}\ e\,x]\!]' \quad = \quad \overline{\lambda}k.\,\underline{\mathsf{let}}\ x = [\![v]\!]_v\ \ \underline{\mathsf{in}}\ \ [\![e\,x]\!]'\,\overline{@}\,k$$

図 9. let 多相の CPS 変換

図 9 からわかるように、let $x = v$ in $e\,x$ の CPS 変換は v と $e\,x$ それぞれについて再帰して CPS 変換を行なっていく。ここで v は図 8 の型付け規則より（CPS 変換前の）inst 関係を受け取らない限り値にはならないことに注意しよう。今、図 9 の右辺は CPS 変換後の項であるため、この Let 項で得られる inst 関係は CPS 変換後のものである。v を値にするためには、この CPS 変換後の inst 関係を何らかの方法で CPS 変換前の inst 関係に戻す必要がある。これができて初めて CPS 変換が可能となる。しかし、3 節のように型を変換前後で異なる形に変換してしまうと、CPS 変換前後で inst 関係の定義も変わることになり、CPS 変換後から前への inst 関係の変換が行うことが困難であることがわかった。

そこで、この節では CPS 変換の前後で項の定義を変更し、CPS 変換後の構文を CPS 変換前の構文と一対一に対応させ、両者で同じ型を使うことで inst 関係の変換を避けることにした。CPS 変換後の項の構文は図 10 のようになる [7]。

$$\begin{array}{rrcl} \text{継続} & k & ::= & x \mid \lambda x.\,e \\ \text{値} & v & ::= & c \mid x \mid \lambda x k.\,e \\ \text{項} & e & ::= & v \mid e_1\,e_2\,k \mid k\,v \mid \mathsf{let}\ x = v\ \mathsf{in}\ e\,x \end{array}$$

図 10. CPS 項

Agda では以下のようになる。cpsvalue[var]τ は τ 型の値を、cpsterm[var]Nat は継続を適用するため最終的に Nat 型となる項を、そして cpscont[var]$\tau \Rightarrow$ Nat は τ 型の項を受け取ったら Nat を返すような継続を示す。継続は CPS 変換を行うことで生み出される項であるため、項や値からは区別して定義する。ラムダ抽象 Fun は CPS 変換を行うと、引数と継続を受け取って、最終的には継続を適用して返す、という形に変換されるため、e_1 はこれまでの引数に加えて、必ず継続を表す var (Typ ($\tau_1 \Rightarrow$ Nat)) 型の変数も受け取るような Agda の関数となっている。関数適用 App は CPS 変換を行うと、関数と引数に加えて継続 c も受け取るようになっている。Let 項については CPS 変換前の定義と同じであるが、最終的に Nat が返るような項となっていることがわかる。このように、項の定義そのものに CPS 変換が反映されているため、CPS 変換前の項と同じ型を扱うことができるようになり、Let 項の CPS 変換を定式化することができるようになった。以後この項を CPS 項と呼び、対応して CPS 変換前の項を DS (Direct Style) 項と呼ぶ。

```
mutual
  data cpscont[_]_⇒Nat {tvar : Set} (var : ts[ tvar ] → Set) : typ[ tvar ] → Set where
    Var : {τ1 : typ[ tvar ]} → (k : var (Typ (τ1 ⇒ Nat))) → -- 継続を表す変数
      cpscont[ var ] τ1 ⇒Nat
    Fun : {τ1 : typ[ tvar ]} → (e1 : var (Typ τ1) → cpsterm[ var ]Nat) → -- 継続を表すラムダ抽象
      cpscont[ var ] τ1 ⇒Nat

  data cpsvalue[_]_ {tvar : Set} (var : ts[ tvar ] → Set) : typ[ tvar ] → Set where
    Var : {σ1 : ts[ tvar ]} → (x : var σ1) → {τ1 : typ[ tvar ]} → inst σ1 τ1 → -- 変数
      cpsvalue[ var ] τ1
    Num : (n : ℕ) → cpsvalue[ var ] Nat -- 自然数
    Fun : {τ1 τ2 : typ[ tvar ]} → -- 継続までを含むラムダ抽象
      (e1 : var (Typ τ2) → var (Typ (τ1 ⇒ Nat)) → cpsterm[ var ]Nat) →
      cpsvalue[ var ] (τ2 ⇒ τ1)

  data cpsterm[_]Nat {tvar : Set} (var : ts[ tvar ] → Set) : Set where
    App : {τ1 τ2 : typ[ tvar ]} → -- 継続の適用までを含む関数適用
      (v1 : cpsvalue[ var ] (τ2 ⇒ τ1)) → (v2 : cpsvalue[ var ] τ2) →
      (c : cpscont[ var ] τ1 ⇒Nat) → cpsterm[ var ]Nat
    Ret : {τ1 : typ[ tvar ]} → -- 継続の適用
      (c : cpscont[ var ] τ1 ⇒Nat) → (v : cpsvalue[ var ] τ1) → cpsterm[ var ]Nat
    Let : (σ1 : ts[ tvar ]) → -- let 多相
      (v1 : {τ1 : typ[ tvar ]} → inst σ1 τ1 → cpsvalue[ var ] τ1) →
      (e2 : var σ1 → cpsterm[ var ]Nat) → cpsterm[ var ]Nat
```

5.4 代入規則と簡約規則

代入規則と簡約規則について定める。DS 項と CPS 項のそれぞれについて代入規則、簡約規則の定義が必要となる。

まず、DS 項についての規則について説明する。代入規則は、型が型スキームに変化するのみで 2 節の定義と基本的には同じである。Let 項の代入規則は、図 11 のようになる。

$$\frac{(\lambda y.\, v_1\, y)[v] \mapsto v_1' \qquad \forall x((\lambda y.\, e_2\, y\, x)[v] \mapsto e_2'\, x)}{(\lambda y.\, (\mathsf{let}\ x = v_1\, y\ \mathsf{in}\ e_2\, y\, x))[v] \mapsto \mathsf{let}\ x = v_1'\ \mathsf{in}\ e_2'\, x}$$

図 11. Let 項の代入規則

Let 項の定義において e_2 は Agda の関数の形をしているため、項の形にするには何らかの引数を与える必要がある。そのため、代入の再帰部分において任意の x を引数として与える形で定義している。

次に DS 項の簡約規則について説明する。簡約規則についても、型が型スキームに変化するのみで 2 節の定義と基本的には同じであるが、Let 項の簡約を追加する必要がある。Let 項の簡約規則は、図 12 のようになる。

$$\frac{(\lambda x.\, e\, x)[v] \mapsto e'}{\mathsf{let}\ x = v\ \mathsf{in}\ e\, x \to e'}$$

図 12. Let 項の簡約規則

$\mathsf{let}\ x = v\ \mathsf{in}\ e\, x$ は、e に含まれる x に v を代入して e' が得られるならば、e' に簡約される。図 12 の規則は 1 ステップ簡約であり、0 ステップ以上の簡約は 2 節と同様に定義する。

次に、CPS 項の代入規則について説明する。CPS 項には、通常の関数適用と継続の適用の 2 種類の関数適用がある。このうち、前者は引数に加えて必ず継続も渡す形になっている。それに合わせて、代入規則も引数と継続両方を代入する規則を定義する。継続も含む代入を以下のように表記する。

$$e[v, c] \mapsto e'$$

これは「e の中の引数部分（最初の引数）に v を代入し、継続部分（ふたつ目の引数）に c を代入したら e' となる」と読む。この代入規則の定義は、代入されるものが二つになるのみで、考え方は 2 節の代入規則と変わらないため、説明は省略する。また、一つの代入のみを扱う代入規則も、継続を扱う際には必要となったため、定義を行なった。これも、2 節と同じ考え方であるため、説明は省略する。

CPS 項の簡約規則を図 13 に示す。簡約規則は、継続付きの β 簡約と継続の適用の 2 種類からなる。CPS 項の簡約規則にはフレームの規則はない。これは、CPS 変換後の項は全てが末尾呼び出しになっており、常に簡約子が一番、外側に現れているからである。

$$\frac{(\lambda x\, k.\, e)[v, c] \mapsto e'}{((\lambda x\, k.\, e)\, v\, c) \to e'} \qquad \frac{(\lambda x.\, e)[v] \mapsto e'}{(\lambda x.\, e)\, v \to e'}$$

図 13. CPS 項の β 簡約規則と継続適用規則

図 13 は 1 ステップの簡約を表す定義であり、0 ステップ以上の簡約規則を定義する必要がある。単純型付き λ 計算において、CPS 変換の正当性の証明は 1 方向のみの複数簡約のみで終了することができるというものであったが、let 多相を加えたところ、より一般的な関係を作らざるを得なくなった。後の、CPS 変換の正当性の証明について説明する際に、両方向の簡約をしなければ証明ができないような状況が出現する。したがって今節では、CPS 項については 0 ステップ以上簡約はどちらの方向の簡約であっても繋がっていれば同値であるという定義になっている。

以後、DS 項の 1 ステップ簡約を $\to$、CPS 項の 0 ステップ以上簡約の同値関係を $\sim$ で表す。

5.5 let 多相を含む単純型付き λ 計算の CPS 変換の正当性の証明

let 多相で拡張した単純型付き λ 計算の CPS の正当性を表す定理は以下のようになる。

定理 3 (CPS 変換の正当性) 任意の DS 項 e, e' について $e \to e'$ が成り立つなら、任意の schematic な継続 κ について $[\![e]\!]\,\overline{@}\,\kappa \sim [\![e']\!]\,\overline{@}\,\kappa$ が成り立つ。

ここで、4 節と同様に schematic な継続というものが出てくるが、定義は 4 節のものとは少々異なるものになっており、定義は以下の通りである。

$$\frac{(\lambda y.\, v_1)[v] \mapsto v_1'}{(\lambda y.\, \kappa\, \overline{@}\, v_1\, y)[v] \mapsto \kappa\, \overline{@}\, v_1'}$$

ある値 v_1 が v の代入により v_1' になるならば、代入前後の値にそれぞれ継続 κ を適用しても代入関係が維持される、という性質を表している。

この schematic の定義で、Let 項以外の項についての正当性の証明を行うことができている。単相の型による単純型付き λ 計算における schematic もこちらの定義にできると考えているが、まだ実際に確認は行っていない。

定理の証明について説明する。4 節の証明に、Let 項の証明を付け加えることになるが、cpsEtaEta′ における Let の場合以外は 4 節と同様に証明が終了する。その際、4 節における補題 eSubst や kSubst を 1 つにまとめたような補題を証明する必要がある。これは、CPS 項の定義により、代入規則を継続の代入も同時に行うように定義したものが存在するため、これに沿った補題が必要となるからである。具体的な証明は、4 節と同様である。2 節と同じような、一つの値の代入に対する補題も、4 節と同様に示す。次に Let 項の簡約の場合については、上の補題と一回の簡約規則を適用することで示すことができた。

次に、cpsEtaEta′ の Let の場合の証明について説明する。ここで、示さなければならないものは以下である。

$$\mathsf{let}\ x = [\![v_1]\!]_v\ \mathsf{in}\ [\![e_2\,x]\!]'\,\overline{@}\,(\lambda v.\,\kappa\,v) \sim \mathsf{let}\ x = [\![v_1]\!]_v\ \mathsf{in}\ [\![e_2\,x]\!]\,\overline{@}\,\kappa$$

両辺 Let 項であるため（手動の証明では）代入によって簡約することができる。このとき、右辺の簡約は 4 節における Reduce*に含まれる簡約とは逆向きの簡約であるため、$\sim$ は双方向の簡約についての同値関係として定義されている。代入による簡約を図 14 に示す。

$$\begin{array}{ll} \text{左辺の簡約} & (\lambda x.\,([\![e_2\,x]\!]'\,\overline{@}\,(\lambda v.\,\kappa\,v)))[[\![v_1]\!]_v] \mapsto [\![e_2']\!]'\,\overline{@}\,(\lambda v.\,\kappa\,v) \\ \text{右辺の簡約} & (\lambda x.\,([\![e_2\,x]\!]\,\overline{@}\,\kappa))[[\![v_1]\!]_v] \mapsto [\![e_2']\!]\,\overline{@}\,\kappa \end{array}$$

図 14. Let 項の簡約

ここで、両辺の簡約結果にある e_2' は、$\lambda x.\,(e_2\,x)[v_1] \mapsto e_2'$ が成り立つような項であり、すなわち e_2 に含まれる x に v_1 を代入した結果の項を表す。両辺の簡約を行なったことで示すべき式は $[\![e_2']\!]'\,\overline{@}\,(\lambda v.\,\kappa\,v) \sim [\![e_2']\!]\,\overline{@}\,\kappa$ となる。これは、cpsEtaEta′ の再帰によって示すことができる。

しかし、Agda による定式化では、e_2' を具体的に与えることができない。代入される項を場合分けすることができれば、それぞれの場合で具体的に e_2' を考えることができるはずだが、代入される項は Agda の λ 式となっている。なぜなら、この定式化は PHOAS によって行われているからである。したがって、Agda において代入される項、$\lambda x.\,(e_2\,x)$ の場合分けを行うことができないため、この項に依存した項である e_2' を具体的なコードとして書くことができないのである。何らかの具体的な引数を与えてやれば λ 式の本体を取り出し場合分けを行うことができるが、今回与えるべき引数の型は、var σ(σ は適当な型スキーム) のようなもので、具体的に作ることは難しい。このため、定理 3 を示すことができなくなってしまった。

図 14 は、単に let 文の progress の性質を述べているにすぎない。しかし、PHOAS を使っているため、それを示すことができず、行き詰っている格好である。これは、PHOAS を使うと多くの証明が簡潔にできている一方で、依然として困難も残っていることを示している。

今のところ、progress を示す方法が見つからないため、本稿ではそれを仮定する形で定式化した。

仮定 1 (Let 項の同値性) 変数を受け取ったら CPS 項を返すような任意の関数 e_1, e_2 について $\forall x(e_1\,x \sim e_2\,x)$ が成り立つなら、任意の CPS 項の値 v について $\mathsf{let}\ x = v\ \mathsf{in}\ e_1\,x \sim \mathsf{let}\ x = v\ \mathsf{in}\ e_2\,x$ が成り立つ。

まず、任意の x のもとで、$e_2\,x$ について cpsEtaEta′ の再帰を行う。すると、任意の x について $[\![e_2\,x]\!]'\,\overline{@}\,(\lambda v.\,\kappa\,v) \sim [\![e_2\,x]\!]\,\overline{@}\,\kappa$ が得られる。この式に対して仮定 1 を適用すると、示したい式が得られる。以上より、仮定 1 を使えば、e_2' を具体的に与えることなく、証明を終了することができる。仮定 1 の証明は、今のところできていない。証明ができない理由は、$\forall x(e_1\,x \sim e_2\,x)$ が Agda において λ 式として表されており、場合分けができないからである。なぜ λ 式を作らなければならないかというと、e_1, e_2 は関数であるため、何らかの引数を与えてやらなければ CPS 項にはならず、$\sim$ 関係を仮定することができないからである。今後、さらに検討するが、これは PHOAS の本質的な難しさにつながっていると思われる。

6 関連研究

CPS 変換の正当性を定式化した研究に、[12] がある。これは定理証明支援系言語 Isabelle/HOL によって FOAS を用いて証明を行っているため、α 変換を全て書き下している。また、[10] では、

対象言語に現れるλ抽象のための de Bruijn index と、CPS 変換時に現れる継続のための de Bruijn index とを分けて取り扱うことでα変換での問題を回避しつつ証明を行っている。

PHOAS に似た変数束縛の表現方法に weak HOAS がある。[14] では、weak HOAS を用いてオブジェクト指向言語である Featherweight Java の定式化を Coq で行っている。PHOAS そのものの応用例に [15] がある。これはグラフの閉路などを取り扱う際、ミュータブルな辺や頂点のリストを使用する従来のアプローチとは異なり PHOAS を用いて表現するものである。

7 まとめと今後の課題

本稿では、単純型付きλ計算における Danvy/Filinski の one-pass の CPS 変換を定式化し、その正当性を Agda で証明した。また、それを let 多相の入った体系においても成り立つようにするため、CPS 項を定義し同様に証明した。いずれの体系においても定式化の方法に PHOAS を採用し、変数束縛はメタ言語の Agda に処理させた。これにより、変数束縛による証明の複雑化を避け、手で書いた証明とほぼ対応した定式化をすることができた。

しかし、let 多相の入った体系では高階関数の中身に立ち入ることができないという問題に直面し、明らかに成り立つと思われる仮定を置く必要があった。この仮定を外せるかどうかの検討は今後の課題だが、PHOAS の本質的な難しさに起因しているのではないかと予想している。

今回、単相の部分については十分に簡潔に CPS 変換の証明を行えることがわかったので、今後、`shift/reset` を加え、selective CPS 変換 [3] の定式化につなげたいと考えている。

謝辞

有益なコメントを下さった査読者の皆様に感謝申し上げます。また、本研究は JSPS 科研費 15K00090 の助成を受けたものです。

参考文献

[1] A. W. Appel. *Compiling with Continuations*. New York: Cambridge University Press, 2007.

[2] K. Asai and Y. Kameyama. Polymorphic Delimited Continuations. *Proceedings of the 5th Asian conference on Programming languages and systems (APLAS'07)*, pp. 239–254, 2007.

[3] K. Asai and C. Uehara. Selective CPS Transformation for Shift and Reset. *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'18)*, pp. 40–52, 2018.

[4] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The POPLMARK Challenge. *Proceedings of the Theorem Proving in Higher Order Logics, 18th International Conference (TPHOLs'05)*, pp. 50–65, 2005.

[5] A. Chlipala. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pp. 143–156, 2008.

[6] A. Chlipala. *Certified Programming with Dependent Types*. Cambridge: MIT Press, 2013.

[7] O. Danvy. Back to Direct Style. *Science of Computer Programming*, Vol. 22, pp. 183–195, 1994.

[8] O. Danvy and A. Filinski. Abstracting Control. *Proceedings of the ACM conference on LISP and functional programming (LFP'90)*, pp. 151–160, 1990.

[9] O. Danvy and A. Filinski. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, Vol. 2, No. 4, pp. 361–391, 1992.

[10] Z. Dargaye and X. Leroy. Mechanized verification of CPS transformations. *Proceedings of the International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'05)*, pp. 211–225, 2007.

[11] J. L. Lawall and O. Danvy. Continuation-based partial evaluation. *Proceedings of the 1994 ACM conference on LISP and functional programming (LFP'94)*, pp. 227–238, 1994.

[12] Y. Minamide and K. Okuma. Verifying CPS transformations in Isabelle/HOL. *Proceedings of the 2003 ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding (MERLIN'03)*, pp. 1–8, 2003.

[13] L. R. Nielsen. A Selective CPS Transformation. *Electronic Notes in Theoretical Computer Science*, Vol. 45, pp. 311–331, 2001.

[14] 奥村健太郎, 五十嵐淳. Weak HOAS を用いた Featherweight Java と Featherweight GJ の Coq 上での形式化. 日本ソフトウェア科学会大会論文集, Vol. 31, pp. 505–518, 2014.

[15] B. C. d. S. Oliveira and W. R. Cook. Functional programming with structured graphs. *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP'12)*, pp. 77–88, 2012.

[16] B. C. Pierce. *Types and Programming Languages*. Cambridge: MIT Press, 2002.

[17] A. Stump. *Verified Functional Programming in Agda*. Morgan & Claypool, 2016.

[18] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and computation*, Vol. 115, No. 1, pp. 38–94, 1994.