Agda による PHOAS を用いた let 多相を含む CPS 変換の正当性の証明

理学専攻・情報科学コース　山田 麗

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 を計算している時の継続は「2∗3 の結果を受け取ったら 1 を足す」となる．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．これを CPS 変換という．CPS 変換を行うとプログラム中の継続を管理することができるので，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．特に，Danvy/Filinski によって定義された one-pass CPS 変換 [7] は，変換時に administrative redex と呼ばれる簡約基を簡約することで変換結果のサイズを小さくすることができる．

本研究では，単純型付きラムダ計算に let 多相を加えた体系に対する one-pass CPS 変換を定義し，その正当性の証明を定理証明系言語 Agda [8] で定式化した．CPS 変換を定式化する際，変数束縛の管理は自明ではない問題である．本研究では PHOAS (Parameterized Higher-Order Abstract Syntax) [4, 3] を用いて束縛変数を管理している．

2　多相の型

多相を含む型の定義は以下である．

$$\begin{array}{lcll} \tau & = & \alpha \mid \mathsf{Nat} \mid \tau \Rightarrow \tau & \text{(型)} \\ \sigma & = & \tau \mid \forall(\hat{\lambda}\alpha.\,\sigma) & \text{(型スキーム)} \end{array}$$

多相の型は単相の型 τ と多相の型スキーム σ によって表される．型 τ は，型変数 α，自然数型 Nat，そして関数型 $\tau \Rightarrow \tau$ によって定義される．型スキーム σ は，単相の型 τ と多相の型 $\forall(\hat{\lambda}\alpha.\,\sigma)$ で定義される．ここで，$\hat{\lambda}\alpha.\,\sigma$ はメタ言語の関数を表していて，本研究においては Agda の関数である．これは PHOAS を用いた定式化によるものである．多相の型は「任意の型変数 α を含むような型スキーム σ」として定義されていて，σ の中で α は任意の型として扱われる．

3　let 多相を含む項

型付きラムダ計算に let 多相を加えた項を定義する．本研究では，項は型を含んで定義されるため，項の定義には型付け規則も含まれる．しかし一般的な型付け規則では，PHOAS を用いた定式化を行うことができない．一般的な let 項の型付け規則は以下である．

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

関数 $\mathit{Gen}(\Gamma, \tau_1)$ とは，Γ に自由に現れない，型 τ_1 に含まれる自由型変数を一般化する関数である．この関数によって，v_1 の型を多相にし，x に割り当てて e_2 の型付けを行うことで let 項の型付けを行なっている．しかし，PHOAS を用いた定式化において変数は全てメタ言語の関数によって束縛されているため，自由変数を表すことはできない．そのため，この型付け規則を定式化することは困難である．そこで，本研究では以下の新しい let 項の型付け規則を扱うことにした．

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

この型付け規則では，まず x に多相の型スキーム σ_1 を割り当てた上で e_2 の型付けを行い，v_1 は σ_1 を具体化して得られる任意の単相の型 τ_1 を持つ，という定義になっている．ここで，$\sigma_1 > \tau_1$ は型スキーム σ_1 を具体化すると型 τ_1 となることを表す関係である．以後，この関係を `Inst` 関係と書く．この型付け規則を用いることで，以下のように項を定義した．

$$\begin{array}{lcll} v & = & x \mid n \mid \lambda\,(\hat{\lambda}x.\,e) & \text{(値)} \\ e & = & v \mid e \mathbin{@} e \mid \mathsf{let}\,(\hat{\lambda}p.\,v)\,(\hat{\lambda}x.\,e) & \text{(項)} \end{array}$$

値 v は，変数 x，数 n，そしてラムダ抽象 $\lambda\,(\hat{\lambda}x.\,e)$ で定義される．ラムダ抽象の束縛変数はメタ言語の関数で管理される．項 e は，値 v，関数適用 $e \mathbin{@} e$，そして let 項 $\mathsf{let}\,(\hat{\lambda}p.\,v)\,(\hat{\lambda}x.\,e)$ で定義される．let 項の本体部分である $\hat{\lambda}x.\,e$ は，束縛変数である x をメタ言語の関数で管理している．値部分である $\hat{\lambda}p.\,v$ はメタ言語の関数を用いているが，PHOAS によるものではなく型付け規則の値部分の型付けにおいて `Inst` 関係を受け取る部分を表しており，つまり p は `Inst` 関係である．このように定義することで，e の中では x は多相に扱われるが v の型は単相に定義することができる．

4　CPS 変換の定義における問題

3 節で定義した項を用いて CPS 変換を定義すると一つ問題が生じる．まず，CPS 変換を行うと型が変化するため，型の CPS 変換を定義しなければならない．型の CPS 変換は $\lfloor\tau\rfloor$ と書く．それを踏まえて CPS 変換の定義を行うと，let 項において型が合わなくなる．let 項の CPS 変換は以下のように定義したい．

$$\begin{array}{lcl} & & \lfloor \mathsf{let}\,(\hat{\lambda}p.\,v_1)\,(\hat{\lambda}x.\,e_2) \rfloor_{\mathsf{S}} \\ & = & \mathsf{let}\,(\hat{\lambda}p'.\,\lfloor(\hat{\lambda}p.\,v_1)\,p'\rfloor_{\mathsf{V}})\,(\hat{\lambda}x.\,\lfloor e_2\rfloor_{\mathsf{D}}\,(\lambda\,(\hat{\lambda}a.\,\kappa\,a))) \end{array}$$

ここで，$\lfloor v\rfloor_{\mathsf{V}}$ は値に対する CPS 変換を，$\lfloor e\rfloor_{\mathsf{S}}$，$\lfloor e\rfloor_{\mathsf{D}}$ は項に対する CPS 変換を表している．項に対する CPS 変換が二種類あるのは Danvy / Filinski の one-pass CPS 変換 [7] による．この変換では，$(\hat{\lambda}p'.\,\lfloor(\hat{\lambda}p.\,v_1)\,p'\rfloor_{\mathsf{V}})$ の部分で型が合わなくなってしまう．理由は，型の CPS 変換である．`Inst` 関係 p は CPS 変換前の let 項のものなので，型は $\sigma_1 > \tau_1$ を持つ．しかし，`Inst` 関係 p' は CPS 変換後の let 項のものなので，型は $\lfloor\sigma_1\rfloor > \tau_1$ となってしまい，$\hat{\lambda}p.\,v_1$ に p' を適用することができない．p' から p を計算することができればこの問題は解決するが，それは単純に求められるものではない．そこで，本研究では型の CPS 変換を避けるため，新しい項を定義することにした．

5 CPS 項

CPS 変換後の項として，CPS 項を以下のように定義する．この項は Danvy による定義 [5] を参考に定義されている．

$$\begin{array}{rcll} c & = & k \mid \lambda^{\mathsf{K}}(\hat{\lambda}x.\, e) & \text{(継続)} \\ v & = & x \mid n \mid \lambda^{\mathsf{C}}(\hat{\lambda}(x,\, k).\, e) & \text{(値)} \\ e & = & v \mid v\,@^{\mathsf{C}}(v, k) \mid k\,@^{\mathsf{K}}\, v \mid & \\ & & \mathsf{let}^{\mathsf{C}}(\hat{\lambda}p.\, v)(\hat{\lambda}x.\, e) & \text{(項)} \end{array}$$

以後，3 節の項を DS 項と書く．DS 項と比べると，CPS 項には継続が増えていることがわかる．継続 c は，継続変数 k，そして継続を表す抽象 $\lambda^{\mathsf{K}}(\hat{\lambda}x.\, e)$ によって定義される．この抽象は引数が一つとなっている．値 v は，変数 x，数 n，そしてラムダ抽象 $\lambda^{\mathsf{C}}(\hat{\lambda}(x,\, k).\, e)$ で定義される．値のラムダ抽象は引数が二つになっていて，これは DS 項のラムダ抽象を CPS 変換することで起こる項の形の変化を項自身の定義に組み込んでいることによる．項 e は，値 v，ラムダ抽象の関数適用 $v\,@^{\mathsf{C}}(v, k)$，継続の関数適用 $k\,@^{\mathsf{K}}\, v$，そして let 項 $\mathsf{let}^{\mathsf{C}}(\hat{\lambda}p.\, v)(\hat{\lambda}x.\, e)$ で定義される．ラムダ抽象は必ず引数を二つ持つように定義されているので，ラムダ抽象の関数適用も二つの引数を適用するような定義となっている．そして，ラムダ抽象の関数適用とは別に，継続の関数適用を定義していて，こちらは引数を一つ受け取るような定義となっている．ラムダ抽象とラムダ抽象の関数適用の型付け規則は以下となる．

$$\frac{\Gamma,\, x : \tau_2,\, k : \tau_1 \Rightarrow \mathsf{Nat} \vdash e : \mathsf{Nat}}{\Gamma \vdash \lambda^{\mathsf{C}}(\hat{\lambda}(x,\, k).\, e) : \tau_2 \Rightarrow \tau_1}$$

$$\frac{\Gamma \vdash v_1 : \tau_2 \Rightarrow \tau_1 \quad \Gamma \vdash v_2 : \tau_2 \quad \Gamma \vdash k_3 : \tau_1 \Rightarrow \mathsf{Nat}}{\Gamma \vdash v_1\,@^{\mathsf{C}}(v_2, k_3) : \mathsf{Nat}}$$

$\lambda^{\mathsf{C}}(\hat{\lambda}(x,\, k).\, e)$ は，二つ引数を受け取るが，最終的には DS 項の関数型と同じ型 $\tau_2 \Rightarrow \tau_1$ で型付けされている．また，$v_1\,@^{\mathsf{C}}(v_2, k_3)$ では，v_1 は CPS 項の関数を表すが，DS 項と同じ関数型 $\tau_2 \Rightarrow \tau_1$ を持つと定義されている．このように CPS 項のラムダ抽象の型付けを行うことで，CPS 変換前後で同じ型を使うことが可能となる．したがって型の CPS 変換も不要となるため，4 節の問題は起こらずに CPS 変換を定義することができる．

6 正当性の証明

let 多相を含むラムダ計算に対する one-pass CPS 変換の正当性の証明を行う．まず，証明にあたって継続に対して以下の制約を設ける必要がある．

定義 1 *static* な継続 κ について，$(\hat{\lambda}y.\, v_1(y))[\lfloor v \rfloor_{\mathsf{V}}] \mapsto v_1'$ を満たす *CPS* 値 v_1，v_1' と *DS* 値 v に対して $(\hat{\lambda}y.\, \kappa\,(v_1(y)))[\lfloor v \rfloor_{\mathsf{V}}] \mapsto \kappa\, v_1'$ を満たすならば κ は *schematic* であるという．

$(\hat{\lambda}y.\, v_1(y))[v] \mapsto v_1'$ は代入を表す関係で，v_1 に含まれる y に v を代入すると v_1' となることを意味する．この定義は「継続 κ が schematic であるというのは，引数の構造を変更しないことである [7]」という意味である．正当性の証明に用いる継続は必ず schematic でなければならない．以下が，証明したい定理である．

定理 2 *DS* 項 e と e' が $e \rightsquigarrow e'$ を満たすならば，*static* で *schematic* な継続 κ について $\lfloor e \rfloor_{\mathsf{S}}\, \kappa \rightsquigarrow \lfloor e' \rfloor_{\mathsf{S}}\, \kappa$ を満たす．

$e \rightsquigarrow e'$ は簡約を表す関係で，e は e' に簡約できることを意味する．定理 2 は，簡約関係にある項は CPS 変換を行なっても簡約関係を維持する，ということを表している．証明は，$e \rightsquigarrow e'$ の導出に関する帰納法で行う．内容は省略するが，二つほどの補題を証明することで，手で証明する場合と同程度の長さで示すことができた．

7 まとめ

Agda で型付きラムダ計算を let 多相で拡張した体系について one-pass CPS 変換を定義し，その正当性を証明した．

変数束縛は PHOAS を用いて管理することで，α 変換などの煩雑な定義を避けることができた．PHOAS を用いて let 項を表現するために，新しく `Inst` 関係を定義した．その結果，CPS 変換を定義する際に型の不一致という問題が発生したが，CPS 変換前後で同じ型を使うための CPS 項を定義することで回避した．

DS 項から CPS 項への one-pass CPS 変換を定義し，正当性を証明した．証明は手で証明を書く場合とほぼ同程度の長さで定式化できた．コードは全部で約 1000 行となっており，これは十分短い定式化であると言える．

今後は，さらに shift/reset [6] を加えた体系に対する selective CPS 変換 [2] の正当性の証明を行いたい．

参考文献

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

[2] 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.

[3] 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, September 2008.

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

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

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

[7] 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.

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