B-021

継続渡しスタイルのラムダ計算のための簡潔な記法に基づくプログラミング言語の提案

A Proposal of a Programming Language based on a Concise Notation for a Lambda Calculus in Continuation Passing Style

島 和之
Kazuyuki Shima

広島市立大学大学院 情報科学研究科
Graduate School of Information Science, Hiroshima City University

1 まえがき

関数からの復帰において計算結果を継続に渡す様式を継続渡しスタイル（Continuation Passing Style, CPS）という [1]．CPS は末尾再帰，コルーチン，例外処理などにおける制御の流れを表現できるので，コンパイラによる最適化や関数の抽出に利用されている [2, 3]．CPS に対し，計算結果を呼び出し元に返す通常の様式を直接スタイル（Direct Style, DS）という．

ラムダ計算は関数の計算を一般的，抽象的に扱う理論であり，プログラミング言語の仕様記述，言語の設計と実装，型システムの研究などに利用されている [4]．DS のラムダ式から CPS のラムダ式を求める CPS 変換の方法が提案されている [5, 6]．ただし，この方法で求められた CPS のラムダ式は DS のラムダ式と比べ，変数と入れ子が多く，複雑になる．しかし，素朴な方法で CPS 変換すると部分適用において合流性を満たさない．

これまでの研究で，CPS のラムダ計算を簡潔に表現し，かつ，部分適用で合流性を満たすことを目的とし，継続を通常の引数と区別する記法を提案した [7]．提案記法では，CPS 用の変換規則を用いることで，合流性を失わずに冗長な変数と入れ子の省略を可能とする．本稿では，DS のラムダ式から提案記法への変換方法，および，提案記法に基づくプログラミング言語を示す．

2 ラムダ式の CPS 変換

DS のラムダ式 M に対する CPS のラムダ式を $[\![M]\!]$ と表記すると，Fischer と Plotkin の CPS 変換は次のように示される [8, 9]．

$$\begin{aligned}
[\![x]\!] &:= \lambda\kappa.\kappa x \\
[\![\lambda x.M]\!] &:= \lambda\kappa.\kappa(\lambda x.[\![M]\!]) \\
[\![MN]\!] &:= \lambda\kappa.[\![M]\!](\lambda\mu.[\![N]\!](\lambda\nu.\mu\nu\kappa))
\end{aligned}$$

ここで x は変数，M, N はラムダ式，κ, μ, ν は DS のラムダ式で未使用の変数とする．

DS のラムダ式と比べ，CPS のラムダ式は変数が $[\![x]\!]$ と $[\![\lambda x.M]\!]$ で 1 つ，$[\![MN]\!]$ で 3 つ増え，入れ子が $[\![\lambda x.M]\!]$ と $[\![MN]\!]$ で 1 つ深くなり，DS の引数が多いほど，CPS はさらに複雑になる．例えば，$0 := \lambda fx.x$, $1 := \lambda fx.fx$ を CPS 変換すると，$[\![0]\!] = \lambda k.k(\lambda cfx.(\lambda k.kx)c)$, $[\![1]\!] = \lambda k.k(\lambda c_1 fx.(\lambda c_2.(\lambda k.kf)(\lambda f.(\lambda k.kx)(\lambda x.fc_2x)))c_1)$ となる．

3 提案記法

提案記法の基本構成要素をハット項（hat terms）という．変数はハット項である．x, c が変数，F, X, C がハット項のとき，$(\hat{}(x)F)$, $(\hat{}cF)$, (FX), $(F.C)$ は，それぞれハット項である．

ハット項の省略形をハット式（hat expressions）という．ハット式の最も外側の括弧は省略し，その他の省略を次のように定義する．

$$\begin{aligned}
\hat{}(x_1x_2\cdots x_n)F &:= \hat{}(x_1)(\hat{}(x_2)(\cdots(\hat{}(x_n)F)\cdots)) \\
\hat{}(x_1x_2\cdots x_n.c)F.C &:= \hat{}(x_1x_2\cdots x_n)(\hat{}cF.C) \\
FX_1X_2\cdots X_n &:= (\cdots((FX_1)X_2)\cdots)X_n \\
F.C &:= (F).C \\
F\hat{}(x_1x_2\cdots x_n)C &:= F.(\hat{}(x_1x_2\cdots x_n)C)
\end{aligned}$$

ここで $x_1, x_2, \cdots, x_n, c$ は変数，$F, X_1, X_2, \cdots, X_n, C$ はハット式，n は自然数とする．

x が変数のとき，x はハット式 x において自由変数である．x が変数，F がハット式のとき，F における自由変数 x はハット式 $\hat{}(x)F$ において束縛変数である．c が変数，F, C がハット式のとき，F, C における自由変数 c はハット式 $\hat{}cF.C$ において束縛変数である．

x が変数，F, X がハット式のとき，F における自由変数 x を X に置換したハット式を $F[x := X]$ と表記すると，ハット式の簡約規則は次のように定義される．

$$\begin{aligned}
\hat{}cF.c &\to F \quad (F \text{ が } c \text{ を含まないとき}) \\
(\hat{}(x)F)X &\to F[x := X] \\
\hat{}c(\hat{}(x)F).C &\to \hat{}c(C(\hat{}(x)F)).c \\
(\hat{}cF.C)X &\to \hat{}\kappa F[c := \hat{}(f)fX].C[c := \hat{}(f)fX] \\
\hat{}c(\hat{}c'F.C').C &\to \hat{}cF[c' := C].C'[c' := C]
\end{aligned}$$

ここで x, c, c' は変数，F, X, C, C' はハット式，κ は F, X, C で未使用，かつ，c 以外の変数とする．

4 ラムダ式からハット式への変換

DS のラムダ式 M に対するハット式を $\langle M \rangle$ と表記すると，次のように定義される．

$$\begin{aligned}
\langle x \rangle &:= x \\
\langle \lambda x.M \rangle &:= \hat{}(x.\kappa)\langle M \rangle.\kappa \\
\langle MN \rangle &:= \hat{}\kappa\langle M \rangle\langle N \rangle.\kappa
\end{aligned}$$

ここで，x は変数，M, N はラムダ式，κ は DS のラムダ式で未使用の変数とする．例えば，チャーチ数は $\langle 0\rangle = \langle \lambda fx.x\rangle = \hat{}(fx.\kappa)x.\kappa$, $\langle 1\rangle = \langle \lambda fx.fx\rangle = \hat{}(fx.\kappa)fx.\kappa$ となる．

合流性は次のように示される．x, y が変数，M, N がラムダ式のとき，$\langle(\lambda xy.M)N\rangle = \hat{}\kappa\langle\lambda xy.M\rangle\langle N\rangle.\kappa = \hat{}\kappa_1(\hat{}(xy.\kappa_2)\langle M\rangle.\kappa_2)\langle N\rangle.\kappa_1 \rightarrow \hat{}\kappa_1(\hat{}(y.\kappa_2)\langle M\rangle[x := \langle N\rangle].\kappa_2).\kappa_1 = \hat{}(y.\kappa)\langle M[x := N]\rangle.\kappa$ である．一方，$(\lambda xy.M)N \rightarrow \lambda y.M[x := N]$, $\langle \lambda y.M[x := N]\rangle = \hat{}(y.\kappa)\langle M[x := N]\rangle.\kappa$ より，合流性を満たす．

例えば，後者関数（チャーチ数 n を渡すと $n+1$ を返す関数）SUCC $:= \lambda nfx.f(nfx)$ にチャーチ数 0 を部分適用すると，$\langle \text{SUCC } 0\rangle = \hat{}\kappa\langle \text{SUCC}\rangle\langle 0\rangle.\kappa = \hat{}\kappa_4(\hat{}(nfx.\kappa_1)f(\hat{}\kappa_2 nfx.\kappa_2).\kappa_1)(\hat{}(fx.\kappa_3)x.\kappa_3).\kappa_4 \rightarrow \hat{}\kappa_4(\hat{}(fx.\kappa_1)f(\hat{}\kappa_2(\hat{}(fx.\kappa_3)x.\kappa_3)fx.\kappa_2).\kappa_1).\kappa_4 \rightarrow \hat{}(fx.\kappa_1)f(\hat{}\kappa_2(\hat{}\kappa_3 x.\kappa_3).\kappa_2).\kappa_1 \rightarrow \hat{}(fx.\kappa)fx.\kappa = \langle 1\rangle$ となり，合流性を満たすことが分かる．

5 提案記法に基づくプログラミング言語

提案記法の応用例を示すため，提案記法に基づくプログラミング言語を提案する．提案言語の実装可能性を示すため，Scheme の処理系 Gauche を用いて提案言語の処理系を開発した．提案言語の構文を EBNF で次に示す．

```
definition='(','define-cps',var,'.',exp,')';
exp=var|hat-exp|lambda-exp;
hat-exp='(','^',param,body,')';
param=var|'(',var,{var},['.',var],')'
body=exp,{exp},['.',exp];
```

`(define-cps var . exp)` は変数 `var` に式 `exp` を束縛する．これは `var` を関数名，`exp` を本体とする関数定義を意味する．S 式の規則により，`(define-cps var exp ···)` は `(define-cps var . (exp ···))` と等しい．`lambda-exp` は Scheme のラムダ式であり，処理系の機能を呼び出すために用いる．

提案記法によって制御の流れを表現できることを示すため，提案言語によるサンプルコードをリスト 1 に示す．その動作を示すため，Scheme で同じ処理を記述したコードをリスト 2 に示す．`main` は 0 から 3 までの整数 `x`，`y` を表示する二重ループの中から，`x+y` が 5 以上のとき，継続 `break` を呼び出す（リスト 1 の 5 行目，リスト 2 の 7 行目）ことで二重ループから脱出する．リスト 1 の 10 行目では，ループを実現するため，不動点コンビネータ `Y` を定義している．`print`，`ifthen`，`+`，`<=`，`>=` の定義は紙面の都合で割愛する．

6 まとめと今後の課題

本稿では，CPS のラムダ計算を簡潔に表現するための記法を提案し，その変換規則を示した．さらに，提案記法に基づくプログラミング言語を示し，制御を表現した例としてループと脱出のサンプルコードを示した．今後の課題としては，提案記法に基づくアクターモデルの表現や並列分散処理への応用などが挙げられる．

リスト 1　提案言語のサンプルコード

```
1  (define-cps main ^(args)
2    Y (^(loop1 x . break)
3        Y (^(loop2 y)
4            print x y ^ c
5            ifthen (>= (+ x y) 5) break ^ c
6            + y 1 ^(y)
7            ifthen (<= y 3)(loop2 y)) 0 ^ c
8        + x 1 ^(x)
9        ifthen (<= x 3)(loop1 x)) 0)
10 (define-cps Y ^(f) f (Y f))
```

リスト 2　Scheme のサンプルコード

```
1  (define (main args)
2    (call/cc
3      (lambda (break)
4        (let loop1 ((x 0))
5          (let loop2 ((y 0))
6            (print x y)
7            (if (>= (+ x y) 5)(break #f))
8            (if (< y 3)(loop2 (+ y 1))))
9          (if (< x 3)(loop1 (+ x 1)))))))
```

参考文献

[1] G.J. Sussman and G.L.S. Jr., “Scheme: A interpreter for extended lambda calculus,” Higher-Order and Symbolic Computation, pp.405–439, Nov. 1998.

[2] 住井英二郎，大根田裕一，米澤明憲，“例外処理機構を備えた命令型言語の CPS 変換とその定式化,” 情報処理学会第 48 回プログラミング研究会，pp.67–82，March 2004.

[3] 廣田知子，浅井健一，“限定継続命令 shift/reset 付き型主導部分評価器の抽出,” 情報処理学会論文誌 プログラミング，vol.6，no.4，pp.50–64，Dec. 2013.

[4] B.C. Pierce, Types and Programming Languages, The MIT Press, 2002.

[5] 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, Dec. 1992.

[6] A. Sabry and M. Felleisen, “Reasoning about programs in continuation-passing style,” LISP and Symbolic Computation: An International Journal, vol.6, pp.289–360, 1993.

[7] 島　和之，“継続渡しスタイルのラムダ計算のための簡潔な記法の提案,” 電子情報通信学会総合大会システム講演論文集 1，vol.D-3-2，p.16，March 2016.

[8] M.J. Fischer, “Lambda-calculus schemata,” LISP and Symbolic Computation: An International Journal, vol.6, pp.259–288, 1993.

[9] G.D. Plotkin, “Call-by-name, call-by-value and the lambda-calculus,” Theoretical Computer Science, vol.1, pp.125–159, 1975.