CB-002

継続渡しスタイルの関数型プログラミング言語における アクターモデルに基づく並行処理の実装

An Implementation of Concurrent Processing based on the Actor Model in a Functional Programming Language in Continuation Passing Style

小島 渚 † 島 和之 ‡
Nagisa Kojima Kazuyuki Shima

† 広島市立大学 情報科学部 システム工学科
Department of Systems Engineering, Faculty of Information Sciences, Hiroshima City University
‡ 広島市立大学大学院 情報科学研究科 システム工学専攻
Department of Systems Engineering, Graduate School of Information Sciences, 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, 2]．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 [3, 4, 5, 6]．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 [7, 8]．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 [9, 10, 11]．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 [12]．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．また，DS のラムダ式から提案記法への変換方法，および，提案記法に基づく関数型プログラミング言語を示した [13]．さらに，提案言語では，参照透過性を保ちつつ破壊的代入のような記述が可能であることを示した [14]．

本稿では，提案言語におけるアクターモデルの実装について述べる．アクターモデルとは，アクター間のメッセージパッシングに基づき，並行システムをモデル化する数学的理論である [15, 16, 17]．一般にメッセージパッシングは同期と非同期に大別されるが，本稿では同期型をさらに 2 つに分け，計 3 種類のメッセージパッシングを提案言語で実装できることを示す．

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 [11, 18]．

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

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，および，そ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可能とし，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．

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

$$
\begin{aligned}
\hat{}c(\hat{}(x)F).C &\rightarrow \hat{}c(C(\hat{}(x)F)).c \\
(\hat{}cF.C)X &\rightarrow \hat{}\kappa F[c := \hat{}(f)fX].C[c := \hat{}(f)fX] \\
\hat{}c(\hat{}c'F.C').C &\rightarrow \hat{}cF[c' := C].C'[c' := C]
\end{aligned}
$$

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 で示す [19].

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

(defineCPS 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 式のドット記法により，(defineCPS var exp ···) は (defineCPS 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　提案言語のサンプルコード

```
1 (defineCPS main ^(args)
2   fix (^(loop1 x . break) x ^(x)
3     fix (^(loop2 y) y ^(y)
4       print x y ^()
5       if (>= (+ x y) 5) break ^()
6       if (< y 3)(loop2 (+ y 1))) 0 ^()
7     if (< x 3)(loop1 (+ x 1))) 0)
```

リスト 2　Scheme のサンプルコード

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

fix，print，+，<=，>=，if の定義をリスト 3 に示す．fix は不動点コンビネータ (fixed-point combinator) である [20]．不動点コンビネータ fix は，任意の f に対し，fix $f = f$(fix f) を満たす高階関数であり，再帰関数の定義に用いられる．リスト 1 の 2 行目の fix から 7 行目の 0 より左までの関数が loop1 に代入される．3 行目の fix から 6 行目の 0 より左までの関数が loop2 に代入される．

6 アクターモデルの実装

アクターモデルでは，アクター間のメッセージパッシングに基づき，並行システムをモデル化する．各アクターは固有のアドレス，メールボックス，振舞を持つ．アクターのアドレスに送信されたメッセージはメールボックスに入れられる．アクターはメールボックスからメッセージを受信したとき，以下の動作からなる振舞を実行できる．

- アクターのアドレスへメッセージを送信する．
- 新しいアクターを生成する．
- 次にメッセージを受信したときの振舞を指定する．

アクターは以下のアドレスを知ることができる．

- 自分自身のアドレス
- 受信したメッセージに含まれたアドレス
- 自分が生成したアクターのアドレス

アクターは既知のアドレスにのみメッセージを送信できる．

一対一のメッセージパッシングは同期メッセージパッシングと非同期メッセージパッシングに大別される．本稿では，同期メッセージパッシングをさらに 2 つに分け，返信同期メッセージパッシングと受領同期メッセージパッシングと名付ける．これら 3 種の違いを図 1 に示す．

リスト 3　その他の関数の定義

```
1  (defineCPS fix ^(f) f (fix f))
2  (defineCPS print lambda args
3    (apply print args))
4  (defineCPS if ^(condition then)
5    condition then ( ) ^(action) action)
6  (defineCPS >= ^(a b) a ^(a) b ^(b)
7    not (< a b))
8  (defineCPS + ^(a b) a ^(a) b ^(b)
9    (lambda (a b)(+ a b)) a b)
10 (defineCPS < ^(a b) a ^(a) b ^(b)
11   (lambda (a b)(< a b)) a b)
12 (defineCPS #t ^(then else) then)
13 (defineCPS #f ^(then else) else)
14 (defineCPS not ^(condition then else)
15   condition else then)
```

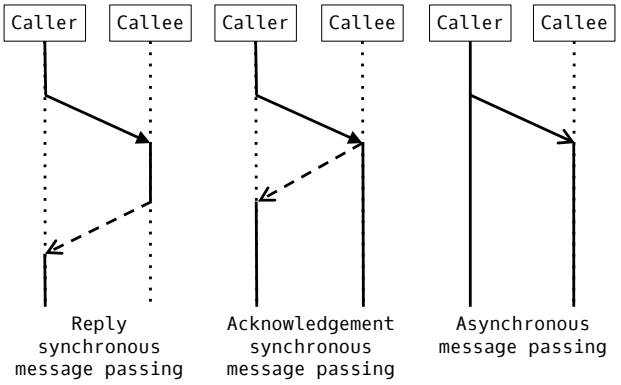

図 1　メッセージパッシングの分類

返信同期メッセージパッシング (reply synchronous message passing) は，呼び出し側 (caller) がメッセージ（実線の矢印）を送信した後，呼び出し先 (callee) から返信（破線の矢印）が届くまで待つ．受領同期メッセージパッシング (acknowledgement synchronous message passing) は，呼び出し側がメッセージを送信した後，呼び出し先から受領通知（破線の矢印）が届くまで待つ．非同期メッセージパッシング (asynchronous message passing) は，呼び出し側がメッセージを送信した後，返信も受領通知も待たずに処理を続行する．

返信同期メッセージパッシングは，オブジェクト指向プログラミングで用いられる．処理の流れが関数呼び出しと同じため，理解し易いが，並行性を表現できない．受領同期メッセージパッシングは，Communicating Sequential Processes (CSP) で用いられる [21]．並行性を表現でき，メッセージのバッファが必要ないが，デッドロックが起こりえる．非同期メッセージパッシングは，アクターモデルで用いられる．並行性を表現でき，ロックの必要はないが，メッセージのバッファが必要となる．

メッセージパッシングのための関数の定義をリスト 4 に示す．sendRplySync（7 行目）は返信同期メッセージパッシング，sendAckSync（12 行目）は受領同期メッセージパッシングでアクターにメッセージを送信する．RplyRqstd（4 行目）は返信を要求するメッセージ，AckRqstd（9 行目）は受領通知を要求するメッセージを生成する．sendSync（1 行目）は返信同期または受領同期でメッセージを送信する．組込み関数 currentActor（2 行目）は現在実行中のアクターを戻り値として返す．sendAsync（3, 6, 8, 11, 13 行目）は非同期メッセージパッシングでアクターにメッセージを送信する．actorNext（6, 11 行目）はアクターに届いた次のメッセージの処理を開始する．start exp（10 行目）は式 exp を並行処理する．

リスト 4　メッセージパッシングのための関数

```
1  (defineCPS sendSync ^(actor message . return)
2    currentActor ^(from)
3    sendAsync actor (message from return) . end)
4  (defineCPS RplyRqstd ^(message from return bhvr)
5    message bhvr ^ r
6    sendAsync from (^(b) actorNext ^( ) r return))
7  (defineCPS sendRplySync ^(actor message)
8    sendSync actor (RplyRqstd message))
9  (defineCPS AckRqstd ^(message from return bhvr)
10   start (message bhvr) ^( )
11   sendAsync from (^(b) actorNext ^( ) return))
12 (defineCPS sendAckSync ^(actor message)
13   sendSync actor (AckRqstd message))
```

リスト 5　メッセージパッシングのサンプルコード

```
1  (defineCPS main ^(args)
2    makeActor (counter 0) ^(a)
3    sendAsync a (^(b) b "Async") ^( )
4    println "main 1" ^( )
5    sendAsync a (^(b) b "Async") ^( )
6    println "main 2" ^( )
7    sendAckSync a (^(b) b "AckSync") ^( )
8    println "main 3" ^( )
9    sendAckSync a (^(b) b "AckSync") ^( )
10   println "main 4" ^( )
11   sendRplySync a (^(b) b "RplySync") ^(r)
12   print "main 5 reply=" r ^( )
13   sendRplySync a (^(b) b "RplySync") ^(r)
14   print "main 6 reply=" r)
15 (defineCPS counter ^(count arg . return)
16   + count 1 ^(count)
17   print arg " Start " count ^( )
18   sleepSec 1 ^( )
19   actorBecome (counter count) ^( )
20   actorNext ^( )
21   sleepSec 1 ^( )
22   print arg " End " count ^( )
23   return count . end)
```

メッセージパッシングのサンプルコードをリスト 5，その実行結果をリスト 6 に示す．makeActor behavior（2 行目）は振舞 behavior を持つアクターを生成し，生成したアクターを戻り値として返す．sleepSec sec（18, 21 行目）は sec 秒の間，実行を中断する．counter count（15 行目）はメッセージを受信する度にカウンターを 1 つずつ増やし，受信したメッセージとカウンターと共に，文字列 Start を表示し，約 2 秒後，文字列 End を表示し，カウンターの値を返すアクターの振舞を示す．

リスト 5 を実行すると main（1 行目）が呼び出される．2 行目で初期値 0 のカウンタを持つアクターを生成し，a に代入する．3, 5 行目でアクター a に文字列 Async を非同期メッセージパッシングで送信する．非同期メッセー

リスト 6　実行結果

```
1  main 1
2  main 2
3  Async Start 1
4  Async Start 2
5  Async End 1
6  AckSync Start 3
7  main 3
8  Async End 2
9  AckSync Start 4
10 main 4
11 AckSync End 3
12 RplySync Start 5
13 AckSync End 4
14 RplySync End 5
15 main 5 reply=5
16 RplySync Start 6
17 RplySync End 6
18 main 6 reply=6
```

ジパッシングなので，アクターの開始（リスト 6 の 3, 4 行目）よりも先に main の処理（リスト 5 の 4, 6 行目）を実行し，リスト 6 の 1, 2 行目が表示されている．リスト 5 の 7, 9 行目でアクター a に文字列 AckSync を受領同期メッセージパッシングで送信する．受領同期メッセージパッシングなので，main の処理（リスト 5 の 8, 10 行目）よりも先にアクターの開始（リスト 6 の 6, 9 行目）が表示されるが，アクターの終了（リスト 6 の 11, 13 行目）は main の処理より後に表示されている．リスト 5 の 11, 13 行目でアクター a に文字列 RplySync を返信同期メッセージパッシングで送信する．返信同期メッセージパッシングなので，アクターの開始（リスト 6 の 12, 16 行目）と終了（リスト 6 の 14, 17 行目）の後，アクターからの戻り値としてカウンタの値を変数 r に代入し，main の処理（リスト 5 の 12, 14 行目）を実行し，リスト 6 の 15, 18 行目でカウンタの値を表示している．

7　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記法，および，それに基づく関数型プログラミング言語を提案した．さらに，提案言語によるアクターモデル，および，メッセージパッシングの実装を示した．提案言語では CPS によって，同期方式に関する 3 種類のメッセージパッシングを簡潔に記述できる．今後の課題としては，提案言語におけるブロック構造の簡略化，モジュール分割，並列分散処理への応用などが挙げられる．

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] J.C. Reynolds, “The discoveries of continuations,” Lisp and Symbolic Computation, vol.6, pp.233–247, Nov. 1993.

[3] 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.

[4] 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.

[5] G.L. Steele, “Rabbit: A compiler for scheme,” Technical Report 474, MIT Artificial Intelligence Laboratory, pp.1–272, May 1978.

[6] C.T. Haynes, D.P. Friedman, and M. Wand, “Continuations and coroutines,” In Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, pp.293–298, Aug. 1984.

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

[8] 小野寛晰，情報科学における論理，日本評論社，1994.

[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, Dec. 1992.

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

[11] M.J. Fischer, “Lambda-calculus schemata,” LISP and Symbolic Computation, vol.6, pp.259–287, Nov. 1993.

[12] 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.

[13] 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に基づくプログラミング言語の提案,” 第 15 回情報科学技術フォーラム (FIT2016)，pp.B–021，Sept. 2016.

[14] 島　和之，“継続渡しスタイルのラムダ計算に基づく破壊的風静的単一代入形式の関数型プログラミング言語,” 第 23 回ソフトウェア工学の基礎ワークショップ (FOSE2016)，pp.127–132，Dec. 2016.

[15] C. Hewitt, P. Bishop, I. Greif, B. Smith, T. Matson, and R. Steiger, “Actor induction and meta-evaluation,” Conference Record of ACM Symposium on Principles of Programming Languages, pp.153–168, Jan. 1974.

[16] C. Hewitt, “Actor model of computation: scalable robust information systems,” arXiv:1008.1459, Aug. 2010.

[17] G.A. Agha, Actors: a model of concurrent computation in distributed systems, Cambridge, Mass. : MIT Press, 1986.

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

[19] ISO and IEC, “Information technology – syntactic metalanguage – extended BNF,” ISO/IEC 14977, International Organization for Standardization, Dec. 1996.

[20] M. Goldberg, “On the recursive enumerability of fixed-point combinators,” BRICS Report RS-05-1, University of Aarhus, 2005.

[21] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall International, 1985.