

# Lisp におけるコード移送機構について\*

渡部卓雄†

北陸先端科学技術大学院大学・情報科学研究科

## Summary

We designed and implemented a Lisp-based scripting language for distributed and mobile computing environment. The language has a simple but generic code mobility mechanism based on first-class partial continuations. When a mobile code — a computing entity that can migrate from host to host — moves, a “slice” of its execution stack should be copied into its destination and then be resumed. Note that the “slice” can be seen as a partial continuation from the migration point. We introduce a language mechanism named `call/ppc` that captures and copies a partial continuation to other places. This mechanism is suitable for Lisp because it can naturally coexist with first-class nested functions. Using this simple mechanism, varieties of code mobility patterns can be described.

## 1 はじめに

本稿では、我々が設計・実装した Lisp(Scheme) ベースのスクリプト言語に導入したコード移送機構を紹介する。このスクリプト言語(名前はまだない)は分散・移動計算機環境における(Lispを用いた)アプリケーション開発を容易にするために設計された。コード移送とは、実行中のプログラムが他の場所(ホスト)に移動し、移動先で実行を再開することである。いいかえると、「ある時点から残りの計算」を抽象化したものである継続(continuation)が、他の場所に転送されることに相当する。実際のコード移送では、転送先からさらに他の場所に移動する(あるいは元の場所に戻る)ことが多い。その場合、それぞれの場所で実質的に用いられるのは、転送された継続の一部だけである。これは部分継続(partial continuation)というものに相当する。本稿で提案するコード移送機構は、(i) 部分継続を具現化し、それを他の場所に転送する関数 `call/ppc` と、(ii) 具現化する部分継続の範囲を指定する構文からなる。この機構を用い

ると、部分継続の転送とその起動(実行の再開)のタイミングを自由に設定でき、様々なコード移送のパターンを記述できる。また、一旦転送した部分継続を再利用することにより、効率的なコード移送が可能になる。

以下、2で提案するコード移送機構について説明し、3でその使用例を示す。4、5で関連研究と課題についてそれぞれ述べる。

## 2 部分継続によるコード移送機構

### 2.1 部分継続

継続(continuation)が、ある時点から「残りすべての計算」を抽象化したものであるのに対し、部分継続(partial continuation)は「ある決った場所までの計算」を抽象化したものである。部分継続をプログラム中で陽に用いることにより、コルーチン等の表現が容易になる。

ここでは部分継続を陽に扱うための言語機構の一つである、一級プロンプト(first-class prompt)[5]を用いて説明する。形式的な定義はA.1にある。まず、Scheme 処理系における一級継続のふるまいについ

\*On Code Mobility Mechanisms in Lisp

†Takuo Watanabe, School of Information Science, Japan Advanced Institute of Science and Technology (JAIST), 1-1 Asahidai, Tatsunokuchi, Ishikawa 923-1292, Japan. Phone: +81-761-51-1256, Facsimile: +81-761-51-1149, E-mail: takuo@acm.org

て考察してみる。多くの Scheme 処理系は対話的に作られているため、あるコード中で継続を呼び出して「残りすべての計算」を終了した後は、処理系のプロンプトがでる。つまり現実の処理系における継続は、つまり、プロンプト(トップレベル)は継続の「終端」としてふるまう。一級プロンプトは、継続が呼び出されたときの終端、つまりどの時点までの部分継続を具現化するかをプログラム中に明示的に指定するものである。

一級プロンプトを # で表し、最も内側の一級プロンプトまでの部分継続を call/pc という関数によって得ることができるとする。次の式

```
(* (# (+ (call/pc
          (lambda (k)
            (k (k 0)))) 1)) 2)
```

では、変数 k に関数として具現化された部分継続がバインドされる。つまり k は (lambda (x) (+ x 1)) に相当する。以下、この部分継続を (+ [] 1) と書く。継続と違い、部分継続は呼び出し元に制御がもどるため、(k (k 0)) のように合成することができる。上の例では、結局 (\* (+ (+ 0 1) 1) 2) を計算していることになる。このように部分継続は関数として合成が可能である。

Scheme では継続は無限エクステントを持った 1 級データであるが、これを実現するには任意の時点におけるコールスタックのコピー(ないしはそれに相当する情報)を保存し、再開時にまたスタックに戻す必要がある。これは一般的に負荷の高い処理である。部分継続の実装上の利点として、スタックのコピーを(最も近い)プロンプトの段階までで抑えられるということが挙げられる。直感的には、一級プロンプトが呼び出された時点から call/pc が呼び出された時点までの制御スタックイメージを「切り出して」保存すればよい [9]。

## 2.2 call/ppc

提案する移動コード機構では、同期型(#)と非同期型(&)の2種類の一級プロンプトと、場所を指定

して部分継続を具現化する call/ppc という関数からなる。形式的な定義は A.2 に与えた。call/ppc は call with placed partial continuation の略である。一級部分継続を具現化する場所を明示的に指定するには、この関数を

```
(call/ppc place function)
```

のように用いることにより、最も近い一級プロンプトから現時点までのスタックイメージと必要なコードが指定された場所 place にコピーされ、引数(function)が呼び出される。

一級部分継続が作られた時の一級プロンプトが同期型である場合、その部分継続を(指定された場所で)呼び出した結果がプロンプトを呼び出した側に伝わる。例えば、場所 A において

```
(# (+1 (call/ppc B
        (lambda (k) (k 2))))))
```

を評価したとする。このときには、場所 B において(+ 1 [])という文脈に相当する部分継続が作成され、これに対する参照が(lambda (k) (k 2))を評価<sup>1</sup>した結果の関数への引数として与えられる。その結果、B において(+ 1 2)が実行された結果の 3 が A に返される。それまでの間、A は同期型一級プロンプトを実行した時点で結果待ちとなる。一方、非同期型の一級プロンプトが最も外側にある式を実行した場合には、結果を待たずに次の式の評価に進む。

例を挙げて説明する。場所 A において、以下の式を実行したとする。

```
(f (# (g (call/ppc B
          (lambda (k1)
            (& (k1
                (h (call/ppc C
                    (lambda (k2)
                      (k2 a))))))))))
```

このとき、部分継続 (g []) が場所 B において生成され、その参照が k1 にバインドされる。さらに

<sup>1</sup>この式自体の評価は A で行われる

部分継続 (k1 (h [])) が場所 C において生成され、その参照が k2 にバインドされる。そして a を評価した結果が k2 に渡される。ここまでは A で実行されることに注意されたい。部分継続 k2 の実行は C で行われる。ここで h の実行結果が k1 に渡され、k1 にバインドされた B 上の部分継続が起動される。この部分継続が作られたときは同期プロンプトを用いていたため、g の結果が A における f に渡される。結局、上の式は (f (g (h a))) と同じであるが、a, h, g, f はそれぞれ A, C, B, A において順に実行されることになる。

### 3 例題

#### 3.1 Go コマンド

Telescript では、実行中のプログラムを他の場所 (プレースというオブジェクトとして抽象化されている) に移送することができる。移送には go というコマンドを用いる。go を実行した時点から先の処理、すなわち継続が他のプレースに渡され、そこで実行されることになる。これを call/ppc を使って書くと以下ようになる。

```
(define (go dest)
  (call/ppc dest
    (lambda (k) (k '()))))
```

これを用いる場合は、以下のように非同期プロンプトで移送範囲を指定する。

```
(& (begin A (go p1) B (go p2) C))
```

#### 3.2 遠隔評価

上の例で & のかわりに # を用いて

```
(# (begin A (go p1) B (go p2) C))
```

とすると、この式を実行した場所では c の計算が終了するまで待つことになり、c の値が式全体の値となる。同様に、場所 place において式 expression

```
(define (who-agent hosts acc k)
  (if (null? hosts)
      (k acc)
      (& (begin
          (go (car hosts))
          (who-agent
            (cdr hosts)
            (cons (exec "who")
                  acc) k))))))

(define (start-agent hosts)
  (# (call/ppc localhost
      (lambda (k)
        (who-agent hosts
          '() k))))))
```

図 1: Round-Trip Agent

を評価する遠隔評価は、以下のように定義することができる。

```
(# (begin (go place) expression))
```

#### 3.3 巡回エージェント

さらに、同期型・非同期型一級プロンプトと call/ppc を用いることにより、一般の同期・非同期型遠隔実行、モバイルエージェント等の様々な遠隔コード実行・モバイルコードを表現することができる。例として、図 1 に簡単なモバイルエージェントの例を示す。例えば

```
(start-agent '("A" "B" "C"))
```

とすることによって、"A", "B", "C" というホストをこの順に巡回し、ログインしているユーザのリストを返す。このプログラムでは最初にエージェントをスタートしたホスト以外では非同期遠隔実行を行い、エージェントがホストを巡回するようになっている。

```
(# (f (call/ppc A (lambda (p) (& (p
  (g (call/ppc B (lambda (q) (& (q
    (h (call/ppc C (lambda (r) (& (r
      a))))))))))))))
```

図 2: Round-Trip Pattern (a)

```
(& (set! v
  (f (call/ppc A (lambda (p) (& (p
    (g (call/ppc B (lambda (q) (& (q
      (h (call/ppc C (lambda (r) (&
        (set! k r))))))))))))))
```

図 3: Round-Trip Pattern (b)

### 3.4 継続の再利用

go では制御が移動する度に継続の移動(コピー)が行われるのに対し, call/cpp では部分継続の移動と制御の移動は独立である. また Scheme における一級継続と同様に, call/cpp によって得られる一級部分継続のエクステン(生存期間)は無限であり, 任意の時点で何度も呼び出すことができる. この性質を用いて, 移動する度に継続をコピーせずに同じ経路を複数回巡回するようなコードを書くことができる. 同じ経路を繰り返し巡回しながらデータを収集するようなアプリケーションを効率良く実現することが可能である.

具体的な記述について述べる. まず, 前節で述べたようにして, 式  $(f (g (h a)))$  を図 2 のように書きかえる. こうすることにより,  $f, g, h$  の本体の実行はそれぞれ場所  $A, B, C$  において行われる. 実行順序としては  $h, g, f$  のようになり, 制御は場所  $C, B, A$  をこの順に巡回することになる.

次に, 図 3 のように最も外側のプロンプトを  $&$  とし,  $r$  を呼び出さずにグローバル変数  $k$  にセットする. この時点では作成された部分継続は実行されない. ここで  $(k a)$  を実行すると部分継続が実行され, 結果が変数  $v$  にセットされる.  $k$  にセットされた部

分継続は, 一度実行された後でもふたたび実行することができる. したがって,

```
(begin (k a) ... (k b) ... (k c) ...)
```

のようなプログラムは

```
(begin (set! v (f (g (h a))))
  ...
  (set! v (f (g (h b))))
  ...
  (set! v (f (g (h c))))
  ...)
```

に相当する. このとき部分継続そのもののコピーは 1 回しか行われていない. このようにして, 同じ経路を複数回通るようなプログラムを比較的効率良く実現することが可能である.

## 4 関連研究

遠隔コード実行や移動コードをサポートする言語はすでにいくつか提案され, いくつかは実用化されている. モーバイルエージェント記述のためのスクリプト言語としては, TeleScript, Agent TCL[7] などがある.

Obliq[2] は, 静的スコープ規則にもとづいた分散オブジェクト指向言語であるが, 任意の手続きクロージャをメッセージに入れて他のノードに送ることが可能であり, 結果としてシンプルな機構で様々な形態の移動コードのプログラミングが可能となる. Obliq のオブジェクトはレコード(名前と値の組の有限個の集まり)として表現され, 内部変数(インスタンス変数)やメソッドは全てレコード要素として扱われる. クラスはなく, あるオブジェクトと類似のオブジェクトは複製操作 (cloning) によって生成する.

Kali-Scheme[4] は Scheme に遠隔実行などの分散計算機構を付加した言語である. サイト間でクロージャや継続を送ることができる機構が用意されている.

go による移動では継続全てのコピーが必要である. これは call/cc と同様に一般に負荷の高い操作である.

移動する範囲を限定するために明示的な部分継続をもちいている例としては [14, 8, 13] などがある。

[1] では移動コードをサポートする関数型言語の操作的意味を与えている。他に移動コードや移動エージェントの形式的意味を与えた例として, [6, 8, 13] がある。また, Cardelli は移動対象や場所を「アンビエント (ambient)」という統一した概念でとらえ, その定式化を行っている [3]。佐藤はアンビエントの考え方にもとづいて, 階層的移動エージェントを提案している [11]。

Servlet に代表される, web サーバ側でプログラムを動かしてサーバの動作をカスタマイズする技術がある。Web サーバで動作する Scheme 処理系 PS<sup>3</sup>P<sup>2</sup> では, ブラウザの back/forward ボタンに連動してプログラムの undo/redo を行うための枠組として継続を用いている。

## 5 議論・課題

最近では, ユーザレベルの移動コードはオブジェクトを移動単位としたいいわゆるモバイルエージェント等のアブストラクションを用いて記述されることが多く, (純粋な) Scheme のような言語に一般的なコード移送機構を実現することはあまりない。本稿で提案している機構は, (オブジェクトを導入していない) Scheme 等で書かれたアプリケーションにモビリティを与えること, およびエージェントのような形式に限らない様々な移動の形態を試してみることを目的としている。我々は Java をベースとした耐故障移動エージェントのためのフレームワークを提案している [15], チェックポイントや再実行などを部分継続を用いて書き直してみるのも興味深い。

我々はこの機構を導入した, Scheme にもとづくスクリプト言語を実装した。現在の実装では, 部分継続が構成されたとき, そこから参照される値はコピーされる。したがって, 移動先で破壊的代入などの副作用を行った結果は移動元には反映されない。また, Oblique のように静的スコープを分散環境に拡張す

<sup>2</sup><http://youpou.lip6.fr/queinnec/VideoC/ps3i.html>

ることはしていない。移動先の情報を積極的に活用することを考えた場合, プログラマに対して動的スコープをうまく使わせるための抽象化機構が重要であると考えられる。また, コピーにあたって様々な Lisp オブジェクトをシリアライズする必要があるが, これは [10] で述べられている方法を用いている。

また, 移動先に作られた部分継続が参照するユーザ定義のデータは全てシリアライズ・コピーしているが, 関数定義などは必要ない場合もある。関口らの JavaGo [12] では, ソースコード変換をすることによって移動させるべきコードとクラスを決定しているが, Scheme のような言語では, これは一般的に静的に判定できない。現在の実装では, 部分継続が作られる際に, シリアライズ機構が参照を辿っている。移動先に効率よくコードを配送するための機構は今後の課題である。

## 参考文献

- [1] R. M. Amadio. Translating core facile. Technical Report ECRC-1994-3, ECRC, 1994.
- [2] L. Cardelli. A language with distributed scope. In *ACM POPL '95*, pp. 286–297, 1995.
- [3] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures*, Vol. 1378 of *Lecture Notes in Computer Science*, pp. 140–155. Springer-Verlag, 1998.
- [4] H. S. Cejtin, S. Jagannathan, and R. Kesley. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, 1995.
- [5] M. Felleisen. The theory and practice of first-class prompts. In *POPL '88*, pp. 180–190. ACM, 1988.
- [6] C. Fournet and G. Gonthier. A calculus of mobile agents. In *CONCUR '96*, Vol. 1119 of *Lec-*

ture Notes in Computer Science, pp. 406–421. Springer-Verlag, 1996.

- [7] R. S. Gray. Agent tcl: A transportable agent system. In *Proc. CIKM 95 Workshop on Intelligent Information Agents*, 1995.
- [8] S. Jagannathan. Communication-passing style for coordination languages. In *Coordination '97*, Vol. 1282 of *Lecture Notes in Computer Science*, pp. 131–149. Springer-Verlag, 1997.
- [9] L. Moreau and C. Queinnec. Partial continuations as the difference of continuations: A duumvirate of control operators. In *PLILP '94*, Vol. 844 of *Lecture Notes in Computer Science*, pp. 182–197. Springer-Verlag, 1994.
- [10] C. Queinnec. Marshaling/demarshaling as a compilation/interpretation process. Research Report LIP6/1998/049, LIP6, 1998.
- [11] I. Satoh. MobileSpaces: A framework for building adaptive distributed applications using a hierarchical mobile agent system. In *ICDCS 2000*, Apr. 2000.
- [12] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension of java language for controllable transparent migration and its portable implementation. In *Coordination '99*, Lecture Notes in Computer Science. Springer-Verlag, 1999.
- [13] T. Sekiguchi and A. Yonezawa. A calculus with code mobility. In *FMOODS '97*, July 1997.
- [14] 渡部, 天野. 移動計算機環境のための遠隔コード実行モデル. 情報処理学会研究報告 (モバイルコンピューティング研究グループ) MBL-3. 情報処理学会, 1996.
- [15] T. Watanabe, N. Amano, and K. Shinbori. Towards a modular substrate for reliable mobile

agent systems. In *Proceedings of ACM/IFIP Middleware 2000 Workshop on Reflective Middleware (RM 2000)*, Apr. 2000.

## A 形式的定義

### A.1 $\lambda_v$ -C-計算

$\lambda_v$ -計算 Felleisen らによって導入された評価文脈 (evaluation context) を用いて値呼び  $\lambda$ -計算 ( $\lambda_v$ -計算) を定義する.  $\lambda_v$ -計算の抽象構文は以下ようになる.

$$\begin{aligned} e & ::= x \mid c \mid \lambda x.e \mid (e e) \\ v & ::= x \mid c \mid \lambda x.e \\ C & ::= [] \mid (C e) \mid (v C) \end{aligned}$$

この  $e$  を  $\lambda_v$ -項と呼ぶ. また  $\lambda_v$ -項のうち, 変数項 ( $x$ ), 定数項 ( $c$ ), ラムダ抽象を値 (value) と呼んで区別し,  $v$  と表記する. さらに,  $C$  は関数適用の項の一部をホール  $[]$  で置きかえたものであり, 評価文脈 (evaluation context) と呼ばれる. 任意の評価文脈  $C$  にはただ 1 つのホールが含まれるが, それを  $\lambda_v$ -項  $e$  で置き換えて作られるものはやはり  $\lambda_v$ -項となる. こうして作られた項を  $C[e]$  と表記する.  $[]$  の置き換えでは, 置き換える項  $e$  の自由変数の束縛回避 (capture avoidance) は行わない.

$\lambda_v$ -計算には  $\delta$  と  $\beta_v$  の 2 種類の簡約がある.  $\beta_v$ -基 ( $\beta_v$ -redex) は  $(\lambda x.e)v$  という形をしている. それぞれの簡約規則を以下のように定義する.

$$C[(f a)] \rightarrow C[\delta(f, a)] \quad (1)$$

$$C[((\lambda x.e) v)] \rightarrow C[e\{v/x\}] \quad (2)$$

$\delta$ -簡約 (1) における  $f, a$  は定数項であり,  $\delta(f, a)$  も  $\delta$ -簡約によって得られた定数項とする. また (2) の  $\beta_v$ -簡約における  $e\{v/x\}$  は,  $e$  中の変数  $x$  の自由な出現を値  $v$  で置き換えたものである.

任意の閉  $\lambda_v$ -項は, 値であるか,  $r$  を簡約基とした  $C[r]$  という形で一意に表現できることが知られている [5]. これによって値呼びによる式評価を表現できる.

$\lambda_v$ -C-計算 部分継続を扱うために、 $\lambda_v$ -計算の構文に  $\#$  と  $\mathcal{F}$  の2つのオペレータを付加する。 $\#$  は一級プロンプトである。

$$e ::= x \mid c \mid \lambda x.e \mid (ee) \mid \mathcal{F}e \mid \#e$$

さらに、以下の2種類の評価文脈を導入する。

$$\begin{aligned} C &::= [] \mid (Ce) \mid (vC) \mid \mathcal{F}C \\ K &::= C \mid K[\#C] \end{aligned}$$

1つ以上の一級プロンプトを含む評価文脈  $K$  を、(一級プロンプトを含まない)評価文脈  $C$  と評価文脈  $K'$  を用いて  $K'[\#C]$  のように書くことができる。このような方法で最も内側の一級プロンプトを明示することができる。

$\lambda_v$ -C-計算では、(1)(2)に加えて以下のような簡約規則がある。

$$C[\mathcal{F}v] \rightarrow (v(\lambda x.C[x])) \quad (3)$$

$$K[\#C[\mathcal{F}v]] \rightarrow K[\#(v(\lambda x.C[x]))] \quad (4)$$

$$K[\#v] \rightarrow K[v] \quad (5)$$

(3)(4)の右辺では、文脈  $C$  を関数  $\lambda x.C[x]$  として具現化していることがわかる。ここで  $x$  は  $C$  中に自由な出現をもたないものとする。なお、一級プロンプトの引数が値になった場合は、一級プロンプトは消滅する(5)。

2.1節で説明した  $\text{call}/\text{pc}$  は、 $\mathcal{F}$  を使って

$$\text{call}/\text{pc} \equiv \lambda f.\mathcal{F}(\lambda k.(f k))$$

のように定義できる。

## A.2 call/ppc

関数  $\text{call}/\text{ppc}$  および一級プロンプト  $\&$ ,  $\#$  の操作的意味を [1] に近い方法で形式的に定義する。最初に  $\lambda_v$ -C-計算にプロセス生成および通信のためのプリミティブを導入した簡単な同期プロセス計算系を定義する。式  $(e)$ , 値  $(v)$ , 文脈  $(C, K)$  はそれぞれ図4のようになる。ここで  $p$  は場所を表す定数項とする。場所  $p$  にある、式  $e$  からなるプロセスを  $\langle e \rangle_p$

と表記する。有限個のプロセスからなる集合をシステムと呼ぶ。

以下、同期プロセス計算系の操作的意味をシステムのリダクションによって定義する。 $\delta/\beta_v$ -簡約によるプロセスの内部状態の変化は以下のようになる。

$$\frac{e \xrightarrow{\delta/\beta_v} e'}{\langle e \rangle_p \rightarrow \langle e' \rangle_p}$$

次にプロセス生成、通信のためのプリミティブ  $\text{spawn}$ ,  $\text{chan}$ ,  $\text{send}$ ,  $\text{recv}$  の意味を以下のように与える。

$$\langle K[\text{spawn}(p', v)] \rangle_p \rightarrow \langle K[p'] \rangle_p, \langle (vp) \rangle_{p'} \quad (6)$$

$$\langle K[\text{chan}(v)] \rangle_p \rightarrow \langle K[(vc)] \rangle_p \quad (7)$$

$$\left. \begin{aligned} \langle K[\text{send}(c, v)] \rangle_p, \\ \langle K'[\text{recv}(c)] \rangle_{p'} \end{aligned} \right\} \rightarrow \langle K[v] \rangle_p, \langle K'[v] \rangle_{p'} \quad (8)$$

(6) は  $\text{spawn}(p', v)$  によって新たに生成されたプロセス  $\langle (vp) \rangle$  が場所  $p'$  に作られることを表している。通信はチャンネルを通して一対一かつ同期的に行われる。(7)のようにチャンネルは  $\text{chan}$  プリミティブによって動的に生成することができる。ここで  $c$  は新しいチャンネルをあらわす fresh な名前(定数)である。こうして得られたチャンネルを使った通信は(8)のようになる。

以上のプリミティブを用いて、一級プロンプトと  $\text{call}/\text{ppc}$  の意味を図5のように定義する。ここで  $e_0; e_1$  は順次実行(Schemeでのbegin)であるが、これは  $e_1$  中で自由な出現を持たない変数  $x$  を用いて  $(\lambda x.e_1)e_0$  と書くことができる。また、右辺に表れるチャンネル名  $c_0, c_1$  は  $\text{chan}$  プリミティブによって得られた fresh なものである。

以上の定義にしたがって、 $\text{call}/\text{ppc}$  を関数として定義することができる。Schemeのシンタクスを用いて定義したものを図6に示す。 $\text{asynch-cont?}$  は引数が非同期の部分継続であるかどうかを調べる関数であるとする。

$$\begin{aligned}
e &::= x \mid c \mid p \mid \lambda x.e \mid (e e) \mid \#e \mid \&e \mid \mathcal{F}e \mid \mathbf{spawn}(e, e) \mid \mathbf{chan}(e) \mid \mathbf{send}(e, e) \mid \mathbf{recv}(e) \mid \\
v &::= x \mid c \mid p \mid \lambda x.e \\
C &::= [] \mid (C e) \mid (v C) \mid \mathcal{F}C \mid \mathbf{spawn}(C, e) \mid \mathbf{spawn}(v, C) \mid \mathbf{chan}(C) \mid \\
&\quad \mathbf{send}(C, e) \mid \mathbf{send}(v, C) \mid \mathbf{recv}(C) \\
K &::= C \mid K[\#C] \mid K[\&C]
\end{aligned}$$

図 4: Syntax

$$\begin{aligned}
\langle K[\&C[(\mathbf{call/ppc} p' v)]] \rangle_p &\rightarrow \langle K[(v (\lambda x.\mathbf{send}(c, x)))] \rangle_p, \langle C[\mathbf{recv}(c)] \rangle_{p'} \\
\langle K[\#C[(\mathbf{call/ppc} p' v)]] \rangle_p &\rightarrow \langle K[(v (\lambda x.\mathbf{send}(c_0, x))); \mathbf{recv}(c_1)] \rangle_p, \langle \mathbf{send}(c_1, C[\mathbf{recv}(c_0)]) \rangle_{p'} \\
\langle C[(\mathbf{call/ppc} p' v)]] \rangle_p &\rightarrow \langle (v (\lambda x.\mathbf{send}(c, x))) \rangle_p, \langle C[\mathbf{recv}(c)] \rangle_{p'}
\end{aligned}$$

図 5: Operational Semantics of **call/ppc**

```

(define (call/ppc p f)
  (call/pc (lambda (k)
    ; 部分継続を得る
    (if (asynch-cont? k)
      ; 部分継続のチェック
      (chan (lambda (c)
        ; 非同期型
        (spawn p (lambda (a) (k (recv c))))
        (f (lambda (x) (send c x))))))
      ; 同期型
      (chan (lambda (c0)
        (chan (lambda (c1)
          (spawn p (lambda (a)
            (send c1 (k (recv c0))))))
          (f (lambda (x) (send c0 x)))
          (recv c1))))))))))

```

図 6: Definition of **call/ppc**