プログラミング 14-11
(1997. 8. 21)

部分継続にもとづく移動コード記述*
— 定義と操作的意味 —

渡部卓雄
北陸先端科学技術大学院大学 情報科学研究科†
takuo@jaist.ac.jp

概要

分散・移動計算機環境における，移動可能プログラムを記述するための一般的な言語機構を提案する．基本的なアイデアは，計算状態の一部を表現する部分継続 (partial continuation) をプログラム中で一級オブジェクトとして明示的に扱うことにある．並行計算系における継続 (continuation) の扱いは一般に繁雑になるが，エクステントを限定した部分継続を得る言語機構を用いることにより，遠隔コード実行の様々なパターンを記述できる．本機構は手続きクロージャ，あるいはそれと等価な機構を持つ様々な逐次言語に導入可能であり，ユーザ向けのモバイルエージェント記述スクリプト言語だけでなく，システムプログラム記述言語にも適用可能である．本稿では操作的意味について延べる．

1 はじめに

プログラムコードを他のサイトに動的に送って実行させる **遠隔コード実行 (remote code execution)**(あるいは移動コード **(mobile code)**) は，次のような理由から分散・移動計算機環境に適している．(a) 物理的制約により十分な計算能力を持たない移動ホストは，(固定) サーバホストにプログラムコードを送り，計算結果を得ることができる．(b) 無線ネットワーク等の低スループットで高レイテンシのネットワークでは，サービスをカスタマイズするためのコードを (固定) サーバ側に送ることによって，インタラクション (パケット往復) の頻度を下げることができる．(c) 移動エージェントを用いて，移動先の様々なサービスを有効に活用することが可能となる．

移動コードのための主な基本技術として，プロセス移送 (process migration) と遠隔評価 (remote evaluation)[11] が挙げられる．一般のプロセス移送や遠隔評価では，プログラムに対して移動を隠蔽することが多いが，移動計算機環境では移動に伴う環境の変化を積極的に活用することが求められる．これは記述言語設計の視点からは，移動に伴う名前空間 (スコープ) や継続 (continuation) の変化をどのようにコントロールするかという問題となる．

本稿では部分継続 **(partial continuation)** を用いるシンプルな遠隔コード実行のモデルを提案する．プログラム中で部分継続を明示的に扱うために，同期型・非同期型の一級プロンプト (first-class prompt) という言語機構を導入した．これによって遠隔コード実行の様々なパターンを記述できる．以下，2 では部分継続について簡単に説明し，3 で部分継続を用いた遠隔コード実行モデルについて説明する．

2 部分継続

2.1 継続

継続 (continuation) は，「ある時点から残りすべての計算」を抽象化した概念である．歴史的には手続き的プログラムの表示的意味を定義する際に導入され，くり返し，ジャンプ，非局所脱出，コルーチンといった様々な制御構造の定義に用いられている．プログラム中で継続を陽に扱うための機構を持つ言語 (例えば Scheme や Standard ML/NJ) では，これらの様々な制御構造をプログラマが定義することができる．

Scheme では `call/cc` という特殊な関数により任意の時点における継続を得ることができる．例えば

*Mobile Code Description using Partial Continuations: Definition and Operational Semantics

†〒923-12 石川県能美郡辰口町旭台 1-1, TEL: 0761-51-1256, FAX: 0761-51-1149

```
(* 2 (call/cc (lambda (k) (+ 3 (k 4)))))
```

という式を評価すると部分式 (call/cc ...) の継続を引数として関数 (ラムダ式) が呼びだされる．結果として (+ 3 (k 4)) の部分式 (k 4) が評価された時点で継続が呼びだされ，式全体の評価結果は 8 となる．

上の例のように，Scheme における継続は関数として具現化 (reify) される．このように，言語の一級 (first-class) データとして表現された継続を**一級継続 (first-class continuation)** と呼ぶ．

2.2 継続の定義

部分継続の概念を導入する前に，文脈書き換え (context rewriting) を用いて一級継続を定義しておく．まず，以下のような抽象構文をもつ値呼びλ-計算 (λ_v-計算) を考える．

$$E \quad ::= \quad x \mid c \mid \lambda x.E \mid EE \qquad (1)$$

ここで E を λ_v-項と呼ぶ．λ_v-項のうち，変数項，定数項，ラムダ抽象を**値 (value)** と呼んで区別し，V と表記する．

$$V \quad ::= \quad x \mid c \mid \lambda x.E \qquad (2)$$

さらに，関数適用の項の一部をホール [] で置きかえたものを**評価文脈 (evaluation context)** と呼び，C と表記する．

$$C \quad ::= \quad [] \mid CE \mid VC \qquad (3)$$

評価文脈 C にはただ1つの [] が含まれるが，それを λ_v-項 e で置き換えて作られるものはやはり λ_v-項となる．こうして作られた項を $C[e]$ と表記する．ここで [] の置き換えは textual に行なわれる．つまりα-変換による e の自由変数の束縛回避 (capture avoidance) は行わない．

λ_v-計算には，δ と β_v の2種類の簡約があり，β_v-基 (redex) は $(\lambda x.E)V$ という形をしている．任意の閉 λ_v-項は値か，さもなくば r を簡約基として $C[r]$ の形で一意に表現されることが知られている．そこで δ-/β_v-簡約を以下のように定義する．

$$\begin{array}{lll} \delta & C[(fa)] & \rightarrow \quad C[\delta(f,a)] \\ \beta_v & C[(\lambda x.E)V] & \rightarrow \quad C[E\{V/x\}] \end{array}$$

ここで f, a は定数項で，$\delta(f,a)$ は δ-簡約によって得られた定数項である．また $E\{V/x\}$ は E 中の x の自由な出現を V で置き換えたものである[1]．以上のような簡約は，Scheme のような値呼びによる式評価を表現している．

次に，λ_v-計算に一級継続を扱うための2種類のオペレータ[2]を付加する．まず λ_v-項と評価文脈の構文 (1)(3) をそれぞれ以下のように拡張する．

$$E \quad ::= \quad x \mid c \mid \lambda x.E \mid \mathbf{abort}\ E \mid \mathbf{call/cc}\ E \qquad (4)$$

$$C \quad ::= \quad [] \mid CE \mid VC \mid \mathbf{abort}\ C \mid \mathbf{call/cc}\ C \qquad (5)$$

オペレータ **abort** は現在の評価文脈を捨てるために用いる．つまり計算を中断して引数 (V) を最終的な結果とする．簡約は以下のように定義される．

$$C[(\mathbf{abort}\ V)] \rightarrow V$$

オペレータ **call/cc** は，現在の評価文脈を関数 $\lambda x.(\mathbf{abort}\ C[x])$ として具現化する．

$$C[(\mathbf{call/cc}\ V)] \rightarrow C[(V\ \lambda x.(\mathbf{abort}\ C[x]))]$$

ただし変数 x は C 中で自由な出現はないものとする．

2.3 部分継続

継続が，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とみなすことができる．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とが知られている．

ここでは，部分継続を陽に扱うための最も単純な言語機構である**一級プロンプト (first-class prompt)**[3] について説明する．まず Scheme 処理系において，一級継続がどのようなふるまいを示すかについて考察してみる．多くの Scheme 処理系はインタラクティブに作られており，継続を呼び出して「残りすべて」の計算を行った後は，処理系のプロンプトがでてくるようになっている．つまり現実の処理系における継続は，ある時点から「処理系のプロンプトが出るまで」の計算を表しているといってよい．つまり，プロンプト (トップレベル) は継続の終端であると考えられる．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

[1] こちらは置き換えによって V 中の自由変数が束縛されないようにしなければならない

[2] 本来，両者とも関数として定義できるが，ここでは簡単にするためオペレータとした．

の一級継続を作成するかをプログラム中に明示的に指定するものである．

次に一級プロンプトを形式的に定義する．(4) の構文に，プロンプト演算子 # によってつくられる項を新たに付加する．また，call/cc のかわりに call/pc を導入する．

$$E \quad ::= \quad x \mid c \mid \lambda x.E \mid \text{abort } E \mid \text{call/pc } E \mid \# \; E$$

さらに，以下の 2 種類の評価文脈を導入する．

$$\begin{array}{lll} C & ::= & [] \mid CE \mid VC \mid \textbf{abort } E \mid \textbf{call/pc } C \\ K & ::= & [] \mid KE \mid VK \mid \\ & & \textbf{abort } E \mid \textbf{call/pc } K \mid \# \; K \end{array}$$

評価文脈 K に 1 つ以上のプロンプト演算子が出現しているとき，これをプロンプト演算子を含まない評価文脈 C を用いて $K'[\# \; C]$ のように書くことができる．

abort と **call/pc** の簡約はそれぞれ以下のように定義される．

$$\begin{array}{lll} C[(\textbf{abort } V)] & \rightarrow & V \\ K[\# \; C[\textbf{abort } V]] & \rightarrow & K[\# \; V] \\ C[\textbf{call/pc } V] & \rightarrow & V \; (\lambda x.C[x]) \\ K[\# \; C[\textbf{call/pc } V]] & \rightarrow & K[\# \; (V \; (\lambda x.C[x]))] \end{array}$$

プロンプトオペレータの引数が値になった場合は，プロンプトオペレータは消滅する．

$$K[\# \; V] \quad \rightarrow \quad K[V]$$

次の例は，ファイル `file` へのストリーム (`stream`) をオープンし，関数 `p` で処理を行い，最後にストリームを閉じるという単純なプログラムである．

```
(let ((stream (open file mode)))
  (# (p stream))
  (close stream))
```

`#` はプロンプトオペレータである．上の式では，(`p stream`) 中で `abort` した場合にはこの時点まで抜けだすことになる．つまり，エラー等で `abort` を使って抜けだしても，ストリームはきちんと閉じられることになる．これは Common-Lisp の `unwind-protect` や Java の例外処理機構 `try-catch` に相当するが，通常の (`call/cc` による) 一級継続で記述するのは困難である．

Scheme `call/cc` のように，継続を無限エクステントを持った 1 級データとして扱うためには，任意の時点における制御スタックのコピー (あるいは，それに相当する情報) を保存し，呼び出し時に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，プロンプトオペレータ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」保存するだけでよい [7]．

一級プロンプトモデルでは，`abort` や `call/pc` のような操作を行った場合に「切り出される」部分継続は，最も近いプロンプトオペレータまでのスタックイメージに相当する．プロンプトオペレータがネストした場合には，どこまでの部分継続が得られるかがプログラマにとって必ずしも明確ではない．この点を改良した部分継続のための言語機構として，shift/reset[2]，splitter[9]，marker-call/pc[7] などが提案されている．

3 遠隔コード実行

本節では，一級プロンプトモデルを拡張した単純な遠隔コード実行モデルについて述べる．ここでは Scheme の構文を用いて説明する．

本モデルでは，同期型 (`#`) と非同期型 (`&`) の 2 種類のプロンプトオペレータ，および `call/pc` のかわりに「場所」を指定して部分継続を作成する `call/ppc`[3] を導入している．それぞれのシンタックスは以下の通りである．

- (`#` *expression*)
- (`&` *expression*)
- (`call/ppc` *place function*)

ここで，*place* はサイト，ホスト等の場所を表現するものであるが，本稿ではモデルの提案だけなので，未定義としておく．

`call/ppc` では，一級部分継続を作る場所を明示的に指定するほかは `call/pc` と同じである．最も近いプロンプトオペレータから現時点までのスタックイメージと必要なコード[4]が指定された場所にコピーされ，引数 (*function*) が呼び出される．

[3] `call-with-placed-partial-continuation` の略
[4] レキシカル変数の値もコピーする必要がある

同期型プロンプトオペレータ　一級部分継続が作られた時のプロンプトオペレータが同期型である場合は，その部分継続が (指定された場所で) 呼び出された結果がプロンプトを呼びだした側に伝わる．例えば，ホスト 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，ホスト B において (+ 1 □) なる部分継続が作成され，これが (lambda (k) (k 2)) を評価[5]した結果の関数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 は同期型プロンプトオペレータを実行した時点で結果待ちとなる．

非同期型プロンプトオペレータ　非同期型のプロンプトオペレータ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．最も近いプロンプトオペレータが非同期であるような部分継続も，結果を返さない．

3.1 操作的意味

関数 call/ppc およびプロンプトオペレータ &, # の意味を形式的に定義する．最初に λ_v 計算に **spawn**, **channel**, **send**, **recv**, というプリミティブを導入し，簡単な同期プロセス計算系を定義する．式 (E)，値 (V)，文脈 (C) はそれぞれ以下のようになる．

$$\begin{aligned} E &::= x \mid c \mid \lambda x.E \mid (E\ E) \mid \mathbf{spawn}(E) \mid \\ &\quad\ \mathbf{chan}(E) \mid \mathbf{send}(E,E) \mid \mathbf{recv}(E) \\ V &::= x \mid c \mid \lambda x.E \\ C &::= [] \mid (C\ E) \mid (V\ C) \mid \mathbf{spawn}(C) \mid \\ &\quad\ \mathbf{chan}(C) \mid \mathbf{send}(C,E) \mid \mathbf{send}(V,C) \end{aligned}$$

式 E を含むプロセスを $\langle E\rangle$ と表記する．有限個のプロセスからなる多重集合 (multiset) Π 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．δ−/β_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．

$$\frac{E \xrightarrow{\lambda_v} E'}{\Pi,\langle E\rangle \rightarrow \Pi,\langle E'\rangle}$$

[5]この式自体の評価はホスト A で行われる

ここで $\Pi,\langle E\rangle$ という記法は$\Pi\cup\{\langle E\rangle\}$ を表している．次に，**spawn** プリミティブの意味は

$$\Pi,\langle C[\mathbf{spawn}(V)]\rangle \rightarrow \Pi,\langle C[\bullet]\rangle,\langle V\ \bullet\rangle$$

のように与えられる．上の•は適当なダミーの定数である．通信はチャネルを通して一対一かつ同期的に行われる．チャネルは **chan** プリミティブによって生成することができる．

$$\Pi,\langle C[\mathbf{chan}(V)]\rangle \rightarrow \Pi,\langle C[V,c]\rangle$$

ここで c は新しいチャネルをあらわす定数である．こうして得られたチャネルを使った通信は

$$\begin{aligned} &\Pi,\langle C[\mathbf{send}(c,V)]\rangle,\langle C'[\mathbf{recv}(c)]\rangle \\ \rightarrow\ &\Pi,\langle C[\bullet]\rangle,\langle C'[V]\rangle \end{aligned}$$

のようになる．

以上のプリミティブを用いて，プロンプトオペレータと call/ppc の意味を以下に定義する．ここでは部分継続を作る場所 (place) については何も延べていないが，新しく生成されたプロセスが存在する場所を p によって指定していると考えればよい．

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

上で $E_0;E_1$ 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 $(\lambda x.E_1)E_0$ のようにすればよい．

この定義にしたがって，call/ppc を Scheme のシンタクスを用いて定義したものを図 1 に示す．ここでは場所を指定してプロセスを生成する spawn-at を使っている．

4 記述例

Telescript では，実行中のプログラムをプレースというオブジェクトで抽象化された場所に移送することができる．移送には go というプリミティブを用いる．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．例えば

```
begin A; go h1; B; go h2; C; end
```

```
(define (call/ppc p f)
  (call/pc (lambda (k)                    ; 部分継続を得る
             (if (asynch-cont? k)         ; 得られた部分継続のチェック
                 (chan (lambda (c)        ; 非同期型の場合
                         (spawn-at p (lambda () (k (recv c))))
                         (f (lambda (x) (send c x)))))
                 (chan (lambda (c0)       ; 同期型の場合．チャネルを2つ生成する．
                         (chan (lambda (c1)
                                 (spawn-at p (lambda () (send c1 (k (recv c0)))))
                                 (f (lambda (x) (send c0 x)))
                                 (recv c1)))))))))
```

図 1: call/ppc の定義

とある場合，B は移送先 h1 で，そして C は h2 でそれぞれ実行される．これを call/ppc を使って書くには，非同期プロンプトを使って以下のように移送先の範囲を指定する．

```
(& (begin A (go h1) B (go h2) C))
```

上で用いた go は，以下のように関数として定義される．

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

上の例で& のかわりに # を使うと，上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．

ここで定義した go を用いて簡単な移動エージェントを記述した例は [12] にある．

5 関連事項

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言語としては，TeleScript，Agent TCL[5]，LogicWare などがある．また，サーバから実行コードをダウンロードする種類の移動コードをサポートする言語には，Java, GNU Guile, CAML, Scheme48 がある．

Obliq[1] 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．

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は [4] や [10] にある．後者はコード移送の問題に関してもある程度言及している．

6 議論・課題

部分継続を用いた遠隔コード実行モデルを提案した．同期・非同期 2 種類のプロンプトオペレータと call/ppc によって，モーバイルエージェント等の様々な遠隔コード実行を表現できる．本モデルでは，遠隔コードの実行環境を部分継続という形で抽象的に表現しているが，これはスタックを用いた (通常のプログラミング言語の) 実行モデルと非常に親和性が高く，効率の良い実装が可能である [7].

しかし本稿で述べているモデルではプロンプトオペレータを用いているため，ネストした場合のふるまいが必ずしも直観的でなく，[2, 9, 7] などで解決しようとしている問題点がそのまま残る[6]．この点を解決するための言語機構の設計は今後の課題である．

また遠隔コード実行では，移動先の名前空間の扱いが問題となる．Obliq[1] のように静的スコープを採用した場合，部分継続の移動先との共有レキシカル変数への破壊的代入などを効率良く実装するのが難しくなる．また，積極的に移動先の情報を用いた

[6] さらに並行性を考慮する必要もある．

い場合には，動的スコープを採用した方がよい場合も多い．GL 言語 [6] のように，部分継続の引数に記憶領域へのポインタを明示的に与えることも検討中である．

ここでは場所を指定して部分継続を作るプリミティブを用いたが，移動先に作られた継続が用いるコードが必ずしも移動先に存在するとは限らない．このモデルを実現する言語が

1. クラスを持つオブジェクト指向言語であり，かつ
2. プロンプトオペレータとそれに対応する `call/ppc` が必ず同じスコープにある

ならば[7]，部分継続に対応するコードをあたかもメソッドのように指定することができ，移動先で用いることができる．もちろんこの場合，移動先に対応するクラスがなければどこかからダウンロードする必要がある．

本方式に基く簡単なコード移送モデルを Java によって実装しているが，現在のところ，移動先の CLASSPATH からコードが得られなければならない．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] Luca Cardelli. A language with distributed scope. In *22nd ACM Symposium on Principles of Programming Languages (POPL '95)*, pp. 286–297, 1995.

[2] Olivier Danvy and Andrezej Fillinski. Representing control: A study of the CPS transformation. Technical Report TR CIS-91-2, Kansas State University, 1992.

[3] Matthias Felleisen. The theory and practice of first-class prompts. In *15th Annual ACM Symposium on Principles of Programming Languages (POPL '88)*, pp. 180–190, 1988.

[4] C. Fournet and G. Gonthier. A Calculus of Mobile Agents. *CONCUR '96*, LNCS 1119, pp. 406-421, 1996.

[5] Robert S. Gray. Agent tcl: A transportable agent system. In James Mayfield and Tim Finin, editors, *Proc. of the CIKM 95 Workshop on Intelligent Information Agents*, December 1995.

[6] Gregory F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a languages and its enviroment. In *15th Annual ACM Symposium on Principles of Programming Languages (POPL '88)*, pp. 158–168, 1988.

[7] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations: A duumvirate of control operators. In *6th Programming Language Implementation and Logic Programming (PLILP '94)*, Vol. 844 of *Lecture Notes in Computer Science*, pp. 182–197, 1994.

[8] 中島達夫，渡部卓雄,「分散オブジェクト技術とモバイルエージェント」，電子通信情報学会誌，1997 年 4 月号.

[9] Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In *18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pp. 174–184, 1991.

[10] Tatsurou Sekiguchi and Akinori Yonezawa. A Calculus with Code Mobility. *Proc. FMOODS '97*, Canterbury, Jul., 1997. (to appear)

[11] J. Stamos and D. Gifford. Remote evaluation. *Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 12, No. 4, pp. 537–565, October 1990.

[12] 渡部卓雄，天野憲樹，部分継続にもとづく移動コード記述，Online Proceedings of WOOC '97, Mar., 1997.

[7] 例えば両者にラベルを付けることで実現できる．