記　号　処　理 75－2
（1994．9．16）

call/cc を用いた繰返し的 Scheme プログラムと CPS 変換

渡辺 佳友　伊藤 貴康

東北大学 情報科学研究科

Scheme は Lisp の方言であり、コンティニュエーションをファーストクラスのオブジェクトとして扱う call/cc を備えた関数型プログラミング言語である。この論文では、Scheme のサブセットである Core Scheme に対して call/cc を用いた繰返し的 Scheme プログラムに基づくコンパイル法を提案する。call/cc を用いた繰返し的 Scheme プログラムとは、再帰的な関数呼出しを let 文と call/cc を用いて末尾再帰風に記述したプログラムである。この論文では、call/cc を用いた繰返し的プログラムによるコンパイルと CPS 法によるコンパイルが、Core Scheme プログラムに対して同じコードを生成することを示す。

Iterative Scheme Programs with call/cc and CPS Conversion

Yoshitomo Watanabe　Takayasu Ito

Department of Computer and Mathematical Sciences
Graduate School of Infomation Sciences
Tohoku University

Scheme, a dialect of Lisp, is a functional programming language. Scheme has a construct **call/cc** to support continuation as first class object. This paper presents two compilation methods of Core Scheme, a subset of Scheme, such that

1) the first method is the CPS method
2) the second method uses a class of Iterative Scheme programs with **call/cc** introduced in this paper

We show that these two methods are equivalent, that is, they produce the same codes for a Core Scheme program. We believe that the method of using Iterative scheme with **call/cc** processes some advantage over the CPS method.

1 はじめに

関数型言語Schemeのコンパイルのためにはスタックによるコンティニュエーション操作を明確にしなくてはならない。CPS(Continuation Passing Style) によるコンパイル法(CPS法)は、コンティニュエーションを関数の引数として明示的に受渡す形式のプログラムにソースプログラムを変換してからコード生成を行うコンパイル法である。CPS法はSteeleによってSchemeのコンパイラRABBIT作成のために採用された。

Schemeはラムダ計算に基づく高階の関数型言語である。Schemeの大きな特徴は、コンティニュエーションをファーストクラスとして扱うための構文**call/cc**を備えていることである。**call/cc**は1引数の関数fを引数に取る。式(**call/cc** f)の評価は、この式のコンティニュエーションをkとすると$(f\ k)$に等しい。

本論文では、Scheme のサブセットである Core Scheme[1]をもとに定義した Core Scheme プログラムのコンパイル法をまず考える。次に、Core Schemeプログラムに対して、**call/cc**を用いた繰返し的Schemeプログラムのクラスを導入する。

繰返し的Schemeプログラムは Core Scheme プログラム実行時のスタック操作を**call/cc**と**let**によって表現したもので、Core Schemeを実現するSchemeのコードとも言える。よって繰返し的SchemeプログラムからCore Scheme プログラムのオブジェクトコードを生成することが可能である(図1参照)。なお、オブジェクトコード生成後の最適化はここでは考えていない。

本論文の構成は以下の通り。2章ではラムダ計算の説明と、スタックマシンモデル,Core Scheme プログラム,そのCPS変換の定義を与える。また、Core SchemeプログラムのCPS法によるコンパイルを示す。3章ではCore Scheme プログラムに対する**call/cc**を用いた繰返し的Scheme プログラムについて説明し、繰返し的Schemeプログラムへの変換規則を定義する。4章で繰返し的Scheme プログラムを用いたコンパイル法を与え、繰返し的Scheme プログラムによるコンパイルはCPS法によるコンパイルと同じコードを生成することを示す。

2 背景

この章では、まずラムダ計算の重要な用語の説明を行った後、文献[1]に従って Core Scheme プログラムとそのCPS変換を与え、スタックマシンモデルを定義し、CPSプログラムからのオブジェクトコード生成規則を与える。

2.1 ラムダ計算

ラムダ計算の式Mは次のBNFによって生成される。

M	$::=$	x	変数
		$\mid \lambda x.M$	λ抽象
		$\mid (M\ M)$	適用

式$\lambda x.M$において変数xは束縛されると言い、そうでない場合自由であると言う。置換とは自由変数を別の式で置き換えることである。$M\{N/x\}$は、式Mにおける変数xの自由な出現を式Nで置換する。

ラムダ計算はα, β, η簡約によって定義されるが、本論文の立場からは次のβ簡約が重要である。

$$\beta\text{簡約}\quad (\lambda x.M)N \rightarrow M\{N/x\}$$

β簡約が可能であるような適用式をβ-redex と呼ぶ。β-redex がないような式をβ標準形と呼ぶ。

2.2 Core Schemeプログラム

本節では Felleisen らの文献[1]で定義されている Core Scheme をもとに Core Scheme プログラムを定義する。

Core Schemeプログラムは関数定義Dの集合である。関数定義を図2の抽象構文で定義する。ここでfが関数名、$\lambda x_1 \ldots x_n.M$が関数本体である。$\lambda x_1 \ldots x_n.M$は実際のプログラムでは`(lambda (x1 ... xn) M)`と記述される。Mは、

- ラムダ式に自由変数は現われない
- ラムダ式は関数として使用される

という2つの条件の下で図2の Core Scheme の BNF に従って生成される式である。Vは変数や値、ラムダ式

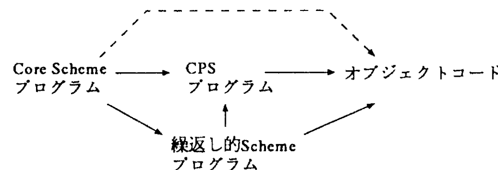

図1：Core Scheme プログラムのコンパイル経路

$$
\begin{array}{lcl}
D & ::= & (\text{define } f\ \ \lambda x_1 \ldots x_n.M) \\
M & ::= & V \\
 & & \mid (\textbf{if } M\ M_1\ M_2) \\
 & & \mid (\textbf{let } (x\ M_1)\ M_2) \\
 & & \mid (O\ M_1\ \ldots\ M_n) \\
 & & \mid (M\ M_1\ \ldots\ M_n) \\
V & ::= & c \mid x \mid \lambda x_1 \ldots x_n.M \\
c & \in & \mathit{Constant} \\
x & \in & \mathit{Variable} \\
O & \in & \mathit{Primitive\ Operator}
\end{array}
$$

図2：Core Scheme プログラムのBNF

などの基本値を表わしている。

式の値を求めることを評価すると言う。式の評価は次のように行われる。if 文 (**if** M M_1 M_2) は Mの値が真ならば M_1を、偽ならば M_2の値を返す。let 文 (**let** (x M_1) M_2) は M_1の値を変数 x に束縛した環境の下での M_2の値を返す。(O M_1 ... M_n) はプリミティブ演算子 Oを引数 M_1 ... M_nの値に適用する。関数適用(M M_1 ... M_n) は引数の値に関数 Mを適用する。

Core Scheme において、ラムダ式$\lambda x_1 \ldots x_n.M$の場合、Mの中の変数 $x_1 \ldots x_n$は束縛されていると言い、let 文 (**let** (x M_1) M_2) の場合、M_2の中の変数 x は束縛されていると言う。束縛されていない変数は自由であると言う。

2.3 スタックマシンモデル

文献[3]をもとにスタックマシンモデルを定義し、Core Scheme に対する CPS プログラムや繰返し的プログラムを実行出来るように、命令とスタックの動作を図3で定義した。

スタックマシンでは、基本演算子は命令 APPLY によって呼び出される。APPLY は演算子と引数の個数を受け取り、スタックトップから n 個の引数に対して演算子の適用を行い、引数をスタックから取り除き、求めた値をスタックに積む。

このスタックマシンにおいて関数適用を実現するには以下のようにコードを生成する。スタックに残りの計算をするコードのラベルを積み、引数をスタックに積んで関数本体のコードにジャンプする。

2.4 Core Scheme の CPS 法によるコンパイル

Felleisen らによる Core Scheme に対する CPS 変換[1]を元に Core Scheme プログラムに対する CPS 変換を定義し、CPS プログラムからのコンパイル法を説明する。

2.4.1 Core Scheme プログラムの CPS 変換

Core Scheme プログラムの関数定義 Dに対する CPS 変換を CPS[D] と表わし、図4で定義する。図4の k はコンティニュエーションを受け取る変数、t は値を受け取る変数である。$\mathcal{F}[\![M]\!]k$または $\mathcal{F}[\![M]\!]\bar{\lambda}t\ldots$は、CPS 変換された式 $\mathcal{F}[\![M]\!]$ のコンティニュエーションが kまたは$\bar{\lambda}t\ldots$ であることを示している。CPS 変換で導入されるラムダ式$\bar{\lambda}\ldots$は、もとからプログラムに存在するラムダ式$\lambda\ldots$と区別される。ラムダ式$\bar{\lambda}\ldots$に関するβ簡約を$\bar{\beta}$簡約と言う。$\bar{\beta}$簡約をしてもプログラムの意味は変わらないので、今後は$\bar{\beta}$標準形を CPS プログラムとして扱う。CPS 変換で導入される変数は、$\bar{\beta}$簡約する際に変数名の衝突が起こらないように選ばれる。$\bar{\beta}$標準形では$\bar{\lambda}$とλを区別しなくてよいので、すべて$\lambda\ldots$と書く。このようにして変換された CPS プログラムの関数定義 D_C の抽象構文は図5(次頁) で示される。

2.4.2 CPS 法によるコンパイル

Core Scheme プログラムの CPS 法によるコンパイルは次の3つの操作からなる。

1. 識別子の renaming
2. CPS 変換
3. コード生成

(APPLY O n)	スタックに積まれている n 個の引数に対して演算子 Oを適用する。引数は消去し結果を積む。
(PUSH-L l)	ラベル lを積む。
(PUSH-C c)	定数 c を積む。
(PUSH-V n)	スタックの n 番目の内容を積む。
(DELETE m n)	スタックの m 番目から n 個を取り除く。
(GOTO l)	ラベル lにジャンプする。
(JUMP-F l)	スタックからポップした値が false ならラベル lにジャンプする。
(RETURN)	スタックの2番目をポップし、そのラベルへジャンプする。

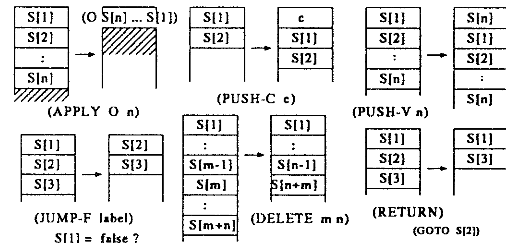

図3：スタックマシンの命令とスタック動作

$$
\begin{aligned}
&\mathrm{CPS}[(\mathbf{define}\ f\ \lambda x_1 \ldots x_n.M)] \\
&\quad = (\mathbf{define}\ f\ \lambda k x_1 \ldots x_n.\mathcal{F}[\![M]\!]k) \\
&\mathcal{F}[\![V]\!] = \bar{\lambda}k.(k\,\Phi[\![V]\!]) \\
&\mathcal{F}[\![(\mathbf{if}\,M_1\ M_2\ M_3)]\!] \\
&\quad = \bar{\lambda}k.(\mathcal{F}[\![M_1]\!]\bar{\lambda}t.(\mathbf{if}\ t\ \mathcal{F}[\![M_2]\!]k\ \mathcal{F}[\![M_3]\!]k)) \\
&\mathcal{F}[\![(\mathbf{let}\,(x\ M_1)\ M_2)]\!] \\
&\quad = \bar{\lambda}k.(\mathcal{F}[\![M_1]\!]\bar{\lambda}t.(\mathbf{let}\ (x\ t)\ \mathcal{F}[\![M_2]\!]k)) \\
&\mathcal{F}[\![(O\ M_1\ \ldots\ M_n)]\!] \\
&\quad = \bar{\lambda}k.(\mathcal{F}[\![M_1]\!]\bar{\lambda}t_1\ \ldots\ (\mathcal{F}[\![M_n]\!]\bar{\lambda}t_n.(O'\ k\ t_1\ \ldots\ t_n))) \\
&\mathcal{F}[\![(M\ M_1\ \ldots\ M_n)]\!] \\
&\quad = \bar{\lambda}k.(\mathcal{F}[\![M]\!]\bar{\lambda}t.(\mathcal{F}[\![M_1]\!]\bar{\lambda}t_1.\ldots \\
&\qquad\qquad (\mathcal{F}[\![M_n]\!]\bar{\lambda}t_n.(t\ k\ t_1\ \ldots\ t_n)))) \\
&\Phi[\![c]\!] = c \\
&\Phi[\![x]\!] = x \\
&\Phi[\![\lambda x_1 \ldots x_n.M]\!] = \lambda k x_1 \ldots x_n.\mathcal{F}[\![M]\!]k
\end{aligned}
$$

図4：Core Scheme プログラムに対する CPS 変換

1. で識別子の renaming をすることにより、CPS 変換が変数のスコープを変化させても混乱することはない。また、関数名も他の識別子と衝突しない。

2. の CPS 変換は前節で定義した通りである。

3. のコード生成は 2 つの処理から成っている。プログラム実行時のスタックの状態をコード生成時に判断するのは困難なので、第 1 段階では次に説明する補助コードを含んだコードを出力する。第 2 段階では、補助コードを削除し、実行時のスタックの情報を所定の位置に書き込む。補助コードは次の 4 つである。

(initialize k $(x_1 \ldots x_n)$)：スタックに入っているラベルと引数を k と $x_1, \ldots, x_n$ に対応させる。

(bind x)：スタックトップにある値を x に束縛して環境を拡張する。

(depth x):スタックにおける変数 x の深さを示す。

(env k)：コンティニュエーション変数 k に関係する環境に束縛される変数の数を示す。

図5 に示す CPS の構文に対して、補助コードを含んだコードを生成する規則 $\mathcal{A}_C$ を図 6 で定義する。

3 call/cc を用いた繰返し的 Scheme プログラム

本章では Core Scheme プログラムに対して、**call/cc** を用いた繰返し的 Scheme プログラム (以下単に "繰返し的 Scheme プログラム" と呼ぶ) を定義し、Core Scheme プログラムからの変換規則を与える。

Scheme での再帰的な関数適用の際には、残りの計算をスタックに積んで関数適用を行わなくてはならない。関数が呼ばれた場所に値を返さなくてよい末尾再帰的な関数適用はスタックに残りの計算を積む必要がない。自分自身を呼出す形式の末尾再帰は自己末尾再帰と呼ばれ、while 文などによる繰返し的な制御を実現する。Scheme においては、自己末尾再帰的に関数を適用することで計算が進むようなプログラムを繰返し的プログラムと言うことが出来る。

$$
\begin{array}{rcl}
D_C & ::= & (\textbf{define}\ f\ \ \lambda k x_1 \ldots x_n . W) \\
W & ::= & (k\ U) \\
 & & \mid (\textbf{if}\ U\ W_1\ W_2) \\
 & & \mid (\textbf{let}\ (x\ U)\ W) \\
 & & \mid (O'\ k\ U_1\ \ldots\ U_n) \\
 & & \mid (O'\ (\lambda x . W)\ U_1\ \ldots\ U_n) \\
 & & \mid (U\ k\ U_1\ \ldots\ U_n) \\
 & & \mid (U\ (\lambda x . W)\ U_1\ \ldots\ U_n) \\
U & ::= & c \mid x \mid \lambda k x_1 \ldots x_n . W \\
O' & \in & \textit{Extended Primitive Operator} \\
k & \in & \textit{Continuation Variable}
\end{array}
$$

図 5：CPS プログラムの BNF

$\mathcal{A}_C[(\textbf{define}\ f\ \lambda k x_1 \ldots x_n . W)]$
$\longrightarrow f:\ \mathcal{A}_C[\lambda k x_1 \ldots x_n . W]$

$\mathcal{A}_C[(k\ U)]$
$\longrightarrow \mathcal{A}_C[U]$
(DELETE 2 (env k))
(RETURN)

$\mathcal{A}_C[(\textbf{if}\ U\ W_1\ W_2)]$
$\longrightarrow \mathcal{A}_C[U]$
(JUMP-F l)
$\mathcal{A}_C[W_1]$
l: $\mathcal{A}_C[W_2]$

$\mathcal{A}_C[(\textbf{let}\ (x\ U)\ W)]$ (U は変数か値)
$\longrightarrow \mathcal{A}_C[U]$
(bind x)
$\mathcal{A}_C[W]$

$\mathcal{A}_C[(\textbf{let}\ (x\ U)\ W)]$ (U はラムダ式)
$\longrightarrow \mathcal{A}_C[W]$
x: $\mathcal{A}_C[U]$

$\mathcal{A}_C[(O'\ k\ U_1\ \ldots\ U_n)]$
$\longrightarrow \mathcal{A}_C[U_1]\ \ldots\ \mathcal{A}_C[U_n]$
(APPLY O n)
(DELETE 2 (env k))
(RETURN)

$\mathcal{A}_C[(O'\ (\lambda x . W)\ U_1\ \ldots\ U_n)]$
$\longrightarrow \mathcal{A}_C[U_1]\ \ldots\ \mathcal{A}_C[U_n]$
(APPLY O n)
(bind x)
$\mathcal{A}_C[W]$

$\mathcal{A}_C[(U\ k\ U_1\ \ldots\ U_n)]$ (U はラベル)
$\longrightarrow \mathcal{A}_C[U_1]\ \ldots\ \mathcal{A}_C[U_n]$
(DELETE $n+1$ (env k))
(GOTO U)

$\mathcal{A}_C[(U\ k\ U_1\ \ldots\ U_n)]$ (U はラムダ式)
$\longrightarrow \mathcal{A}_C[U_1]\ \ldots\ \mathcal{A}_C[U_n]$
(DELETE $n+1$ (env k))
$\mathcal{A}_C[U]$

$\mathcal{A}_C[(U\ (\lambda x . W)\ U_1\ \ldots\ U_n)]$
$\longrightarrow$ (PUSH-L l) (U はラベル)
$\mathcal{A}_C[U_1]\ \ldots\ \mathcal{A}_C[U_n]$
(GOTO U)
l: (bind x)
$\mathcal{A}_C[W]$

$\mathcal{A}_C[(U\ (\lambda x . W)\ U_1\ \ldots\ U_n)]$
$\longrightarrow$ (PUSH-L l) (U はラムダ式)
$\mathcal{A}_C[U_1]\ \ldots\ \mathcal{A}_C[U_n]$
$\mathcal{A}_C[U]$
l: (bind x)
$\mathcal{A}_C[W]$

$\mathcal{A}_C[c] \longrightarrow$ (PUSH-C c)

$\mathcal{A}_C[x] \longrightarrow$ (PUSH-C (depth x))

$\mathcal{A}_C[\lambda k x_1 \ldots x_n . W]$
$\longrightarrow$ (initialize k $(x_1\ \ldots\ x_n)$)
$\mathcal{A}_C[W]$

図 6：CPS プログラムからのコード生成規則 $\mathcal{A}_C$

call/cc を用いた繰返し的 Scheme プログラムとは、この考えを拡張して、再帰的な関数適用を

1) コンティニュエーション (call/cc)

2)let 構文

によって末尾再帰風に表現した Scheme プログラムである。

繰返し的 Scheme プログラムでは let 文によってプログラムのコンティニュエーションを明確に表現し、再帰的な関数のコンティニュエーションを call/cc によって変数に束縛することによって、再帰的関数の制御を行う。繰返し的 Scheme プログラムはスタックによる実行制御を let と call/cc を用いて記述している。

3.1 繰返し的プログラムの具体例による説明

まず繰返し的 Scheme プログラムの考え方を例を用いて具体的に説明する。Core Scheme プログラムの式は inside-out と left-to-right のスタイルで評価される。Core Scheme の式 `(- (+ a b) (+ c d))` を繰返し的スタイルで記述すると以下のようになる。

```
(let (x (+ a b))
  (let (y (+ c d))
    (let (z (- x y))
      (k z))))
```

式を評価しその結果を変数に束縛する逐次制御を let 文で記述することによって、Core Scheme の式の評価順を明確に示している。k が繰返し的スタイルの式における最初の let 構文の評価に対するコンティニュエーションを示しているとすると、その適用 `(k z)` は Core Scheme の式 `(- (+ a b) (+ c d))` の値をスタックに積んで次の計算にプログラム制御を移すことに相当する。

次に Core Scheme プログラムの関数定義に相当する繰返し的 Scheme プログラムの関数定義の考え方を例によって具体的に説明する。階乗を求める Core Scheme プログラムの関数

```
(define fact (lambda (n)
               (if (= n 1)
                   1
                   (* n (fact (- n 1))))))
```

を繰返し的 Scheme プログラムの関数として以下のように記述する。ただし Core Scheme プログラムの `fact` と区別するために、繰返し的 Scheme プログラムの階乗を求める関数を `fact-i` と表記し、以下この関数名で両者を区別する。

```
(define fact-i
        (lambda (k n)
          (let (t1 (= n 1))
            (if t1
                (k 1)
                (let (t2 (- n 1))
                  (let (t3 (call/cc
                             (lambda (k')
                               (fact-i k' t2))))
                    (let (t4 (* n t3))
                      (k t4))))))))
```

関数 `fact` のスタックによる実行制御を、繰返し的 Scheme プログラムの関数 `fact-i` がどのように実現するのかを以下の 6 点で説明する。

1. 関数 `fact` を実行する際には、呼び出された時点に値を返すという次の計算はスタックに積まれている。`fact-i` を実行する際には、`fact-i` のコンティニュエーションを `k` に受け取る。

2. 関数 `fact` が if 文による分岐をするコードをスタックに積んで条件部 `(= n 1)` の値を求めることを、`fact-i` は、if 文を本体に持つ let 文の初期化部分で `(= n 1)` の値を求めることによって表現する。

3. `fact` が `(= n 1)` の値をスタックに積み、その値で条件分岐することを、`fact-i` は `(= n 1)` の値を let 構文で変数 `t1` に束縛し、`t1` の値で条件分岐することによって表現する。

4. `fact` が 1 をスタックに積んで自分を呼んだコードに制御を移すことを、`fact-i` はコンティニュエーション `k` を 1 に適用することによって表現する。

5. `fact` が `(* n (fact (- n 1)))` における再帰的な `fact` の呼出しのために残りの計算をスタックに積み、さらに `fact` の引数 `(- n 1)` の評価を先に行うことを、`fact-i` は `fact` の再帰的呼出しを実現する let 文を本体とする let 文の初期化部分で `(- n 1)` を評価することによって表現する。

6. `fact` が乗算をするコードをスタックに積んで `fact` を再帰的に呼出すことを、`fact-i` は、乗算をするコードを本体とする let 文の初期化部分で、call/cc を用いて `k'` に束縛したコンティニュエーションを引数として末尾再帰風に `fact-i` を呼出すことによって表現する。

3.2 繰返し的 Scheme プログラムの定義

Core Scheme プログラムにおける関数定義 D を繰返し的 Scheme プログラムに変換する規則 $IS[D]$ を図 8 に定義し、簡単な説明を与える。

- 関数定義：識別子 define と関数名は何も変換しない。関数本体であるラムダ式は、その関数が適用される時のコンティニュエーションを受け取るように仮引数を 1 つ増やし、この仮引数をラムダ式の本体のコンティニュエーションとみなして本体を変換する。
- 値：与えられたコンティニュエーションを適用するように変換する。
- if 文：条件部を先に評価することを let 文で示すように変換する。let 文の本体は、条件部の値が束縛される変数で分岐する if 文とする。

● let 文：初期化部分を先に評価することを新しい let 文で示すように変換する。その値を変数に束縛することを示す let 文を新しい let 文の本体とする。

● 基本演算子適用：引数の評価順を let 文によって明確にする。引数の評価が終了してから演算子の適用を let 文の中で行う。

● 関数適用：再帰的適用ならばコンティニュエーションを let の本体として明確に表現する。このコンティニュエーションを call/cc で捉え、末尾再帰的な構文に変換された関数に第1引数として渡す。末尾再帰的適用ならば、引数を評価した後に、与えられたコンティニュエーションをそのまま引数として受け取って適用するように変換する。

規則 I_k における k はコンティニュエーション変数を示す。I_k によって生成された let 文の初期化部分を変換するのが規則 L である。L が変換する let 文の初期化部分が値であるような場合、値評価に対してのコンティニュエーションは明確でなくてもよいので、$L[(\textbf{let}\ (s\ V)\ T)] = (\overline{\lambda} s.T)J[V]$ と変換することで、余分な let 文を削除する。

Core Scheme のプログラムが図8の規則によって変換されて繰返し的プログラムになると、その関数定義 D_I は以下の条件を満たし、かつ図7の抽象構文に従う。

1. 式の評価は let 文の初期化部分で行われ、関数の引数はすべて値であることから、コンティニュエーションがその let 文の本体で表現されている。
2. 再帰的な関数適用の残りの計算は call/cc で変数に束縛されて関数に渡される。関数適用の構文は末尾再帰風になっている。
3. 関数は明示的なコンティニュエーション適用によって値を返す。

4 繰返し Scheme プログラムからのコンパイル

本章では、call/cc を用いた繰返し的 Scheme プログラム (以下単に "繰返し的 Scheme プログラムと呼ぶ) からのコンパイル法を説明する。Core Scheme プログラムを繰返し的 Scheme プログラムに変換し、この繰返し的 Scheme プログラムからのコード生成が、Core Scheme プログラムの CPS 法によるコンパイルと同じコードを生成することを証明する。

4.1 コンパイル法の定義

繰返し的 Scheme プログラムを用いた Core Scheme プログラムのコンパイルは次のように行われる。

$$
\begin{array}{rcl}
D_I & ::= & (\textbf{define}\ f\ \lambda k x_1 \ldots x_n.T) \\
T & ::= & (k\ S) \\
 & & \mid (\textbf{if}\ S\ T_1\ T_2) \\
 & & \mid (\textbf{let}\ (x\ S)\ T) \\
 & & \mid (\textbf{let}\ (x\ (O\ S_1\ \ldots\ S_n))\ T) \\
 & & \mid (\textbf{let}\ (x\ (\textbf{call/cc} \\
 & & \qquad \lambda k.(S\ k\ S_1\ \ldots\ S_n)))\ T) \\
 & & \mid (S\ k\ S_1\ \ldots\ S_n) \\
S & ::= & c \mid x \mid \lambda k x_1 \ldots x_n.T \\
k & \in & \textit{Continuation Variable}
\end{array}
$$

図7：繰返し的 Scheme プログラムの BNF

$$
\begin{array}{rcl}
\mathrm{IS}[(\textbf{define}\ f\ \lambda x_1 \ldots x_n.M)] & = & (\textbf{define}\ f\ \lambda k x_1 \ldots x_n.\mathrm{I}_k[M]) \\
\mathrm{I}_k[V] & = & (k\ \mathrm{J}[V]) \\
\mathrm{I}_k[(\textbf{if}\ M\ M_1\ M_2)] & = & \mathrm{L}[(\textbf{let}\ (s\ M)\ (\textbf{if}\ s\ \mathrm{I}_k[M_1]\ \mathrm{I}_k[M_2]))] \\
\mathrm{I}_k[(\textbf{let}\ (x\ M_1)\ M_2)] & = & \mathrm{L}[(\textbf{let}\ (s\ M_1)\ (\textbf{let}\ (x\ s)\ \mathrm{I}_k[M_2]))] \\
\mathrm{I}_k[(O\ M_1\ \ldots\ M_n)] & = & \mathrm{L}[(\textbf{let}\ (s\ (O\ M_1\ \ldots\ M_n))\ \mathrm{I}_k[s])] \\
\mathrm{I}_k[(M\ M_1\ \ldots\ M_n)] & = & \mathrm{L}[(\textbf{let}\ (s_0\ M)\ \mathrm{L}[(\textbf{let}\ (s_1\ M_1)\ \ldots \\
 & & \quad \mathrm{L}[(\textbf{let}\ (s_n\ M_n)\ (s_0\ k\ s_1\ \ldots\ s_n))]\ldots)])] \\
\mathrm{J}[c] & = & c \\
\mathrm{J}[x] & = & x \\
\mathrm{J}[\lambda x_1 \ldots x_n.M] & = & \lambda k x_1 \ldots x_n.\mathrm{I}_k[M] \\
\mathrm{L}[(\textbf{let}\ (s\ V)\ T)] & = & (\lambda s.T)\ \mathrm{J}[V] \\
\mathrm{L}[(\textbf{let}\ (s\ (\textbf{if}\ M\ M_1\ M_2))\ T)] & = & \mathrm{L}[(\textbf{let}\ (s'\ M)\ (\textbf{if}\ s'\ \mathrm{L}[(\textbf{let}\ (s\ M_1)\ T)]\ \mathrm{L}[(\textbf{let}\ (s\ M_2)\ T)]))] \\
\mathrm{L}[(\textbf{let}\ (s\ (\textbf{let}\ (x\ M_1)\ M_2))\ T)] & = & \mathrm{L}[(\textbf{let}\ (s'\ M_1)\ (\textbf{let}\ (x\ s')\ \mathrm{L}[(\textbf{let}\ (s\ M_2)\ T)]))] \\
\mathrm{L}[(\textbf{let}\ (s\ (O\ M_1\ \ldots\ M_n))\ T)] & = & \mathrm{L}[(\textbf{let}\ (s_1\ M_1)\ \ldots\ \mathrm{L}[(\textbf{let}\ (s_n\ M_n) \\
 & & \quad (\textbf{let}\ (s\ (O\ s_1\ \ldots\ s_n))\ T))]\ldots)] \\
\mathrm{L}[(\textbf{let}\ (s\ (M\ M_1\ \ldots\ M_n))\ T)] & = & \mathrm{L}[(\textbf{let}\ (s_0\ M)\ \mathrm{L}[(\textbf{let}\ (s_1\ M_1)\ \ldots\ \mathrm{L}[(\textbf{let}\ (s_n\ M_n) \\
 & & \quad (\textbf{let}\ (s\ (\textbf{call/cc}\ \lambda k.(s_0\ k\ s_1\ \ldots\ s_n)))\ T))]\ldots)])]
\end{array}
$$

図8：Core Scheme プログラムから繰返し的プログラムへの変換規則

1. 識別子の renaming
2. 繰返し的 Scheme プログラムへの変換
3. コード生成

1. は CPS 法と同じである。2. については前章で述べた。3. のコード生成は CPS 法と同様に 2 段階に分かれている。第 1 段階により、4 頁に示した補助コードを含むオブジェクトコードが生成される。第 2 段階ではこのオブジェクトコードの補助コードを削除し、適切なスタックの情報を書き込む。この第 2 段階の処理は CPS 法と同じとする。従って、繰返し的 Scheme プログラムの関数定義に対して補助コードを含んだオブジェクトコードを生成する規則 $\mathcal{A}_I$ を図 9 で定義する。

4.2 CPS 法との比較

Core Scheme プログラムの 2 つの方法によるコンパイル、つまり **call/cc** を用いた繰返し的 Scheme プログラムによるコンパイルと CPS 法によるコンパイルは同じコードを生成することが出来る。本節ではその証明の考え方を説明する。

話を単純にするため識別子の renaming は考えない。2 つのコンパイル法によるコードが等しいことを証明するためには、Core Scheme プログラムの関数定義 D を繰返し的 Scheme プログラムに変換して生成したオブジェクトコード $\mathcal{A}_I[\mathrm{IS}[D]]$ と、CPS プログラムに変換して生成したオブジェクトコード $\mathcal{A}_C[\mathrm{CPS}[D]]$ に対し

$$\mathcal{A}_I[\mathrm{IS}[D]] \equiv \mathcal{A}_C[\mathrm{CPS}[D]]$$

を示せばよい。≡は両辺のコードが等しいことを示している。

変換 IS をプログラムに適用することを $\xrightarrow{\mathrm{IS}}$ と表わすと、Core Scheme プログラムの任意の関数定義

$$(\textbf{define}\ f\ \ \lambda x_1 \ldots x_n.M)$$

に対して両辺が生成するオブジェクトコードは次のよ

$(\textbf{define}\ f\ \ \lambda k x_1 \ldots x_n.T)$: 関数名 f をラベルとして、その行から関数本体のコードを開始する。

$\mathcal{A}_I[(\textbf{define}\ \ f\ \ \lambda k x_1 \ldots x_n.W)]$
$\longrightarrow f:\ \mathcal{A}_I[\lambda k x_1 \ldots x_n.W]$

$(k\ S)$: 基本値 S の値をスタックに積み、環境を捨て、ラベルが示す番地にジャンプする。

$\mathcal{A}_I[(k\ S)] \ \longrightarrow\ \mathcal{A}_I[S]$
(DELETE 2 (env k))
(RETURN)

$(\textbf{if}\ S\ T_1\ T_2)$: S の値をスタックに積み、分岐する。

$\mathcal{A}_I[(\textbf{if}\ S\ T_1\ T_2)] \ \longrightarrow\ \mathcal{A}_I[S]$
(JUMP-F l)
$\mathcal{A}_I[T_1]$
$l:\ \mathcal{A}_I[T_2]$

$(\textbf{let}\ (x\ S)\ T)$: S は変数か定数の時。S の値を積み、その値を (bind x) で束縛して T のコードを生成する。

$\mathcal{A}_I[(\textbf{let}\ (x\ S)\ T)] \ \longrightarrow\ \mathcal{A}_I[S]$
(bind x)
$\mathcal{A}_I[T]$

S がラムダ式の時。S のコードのラベルを l として T のコードを生成する。

$\mathcal{A}_I[(\textbf{let}\ (x\ S)\ T)] \ \longrightarrow\ \mathcal{A}_I[T]$
$x:\ \mathcal{A}_I[S]$

$(\textbf{let}\ (x\ (O\ S_1\ \ldots\ S_n))\ T)$: T が $(k\ x)$ の時。$S_1, \ldots, S_n$ の値を積み、演算子を適用し、ラベルの示す番地に返る。

$\mathcal{A}_I[(\textbf{let}\ (x\ (O\ S_1\ \ldots\ S_n))\ (k\ x))]$
$\longrightarrow \mathcal{A}_I[S_1]\ \ldots\ \mathcal{A}_I[S_n]$
(APPLY O n)
(DELETE (env k))
(RETURN)

T が $(k\ x)$ でない時。$S_1, \ldots, S_n$ の値を積み、演算子を適用し、結果を x に束縛して T を評価する。

$\mathcal{A}_I[(\textbf{let}\ (x\ (O\ S_1\ \ldots\ S_n))\ T)]$
$\longrightarrow \mathcal{A}_I[S_1]\ \ldots\ \mathcal{A}_I[S_n]$
(APPLY O n)
(bind x)
$\mathcal{A}_I[T]$

$(S\ k\ S_1\ \ldots\ S_n)$: S がラベルの時。引数の値を積んだ後に環境を捨て、S へジャンプする。

$\mathcal{A}_I[(S\ k\ S_1\ \ldots\ S_n)]$
$\longrightarrow \mathcal{A}_I[S_1]\ \ldots\ \mathcal{A}_I[S_n]$
(DELETE $n+1$ (env k))
(GOTO S)

S がラムダ式の時。関数適用は S のコード実行である。

$\mathcal{A}_I[(S\ k\ S_1\ \ldots\ S_n)]$
$\longrightarrow \mathcal{A}_I[S_1]\ \ldots\ \mathcal{A}_I[S_n]$
(DELETE $n+1$ (env k))
$\mathcal{A}_I[U]$

$(\textbf{let}\ (x\ (\textbf{call/cc}\ \lambda k.(S\ k\ S_1\ \ldots\ S_n)))\ T)$: S がラベルの時。x に値を束縛して T_2 を実行するコードの前にラベル l を付ける。このラベルをスタックに積み、引数をスタックに積んでラベル S にジャンプする。

$\mathcal{A}_I[(\textbf{let}\ (x\ (\textbf{call/cc}\ \lambda k.(S\ k\ S_1\ \ldots\ S_n)))\ T)]$
$\longrightarrow$ (PUSH-L l)
$\mathcal{A}_I[S_1]\ \ldots\ \mathcal{A}_I[S_n]$
(GOTO S)
$l:$ (bind x)
$\mathcal{A}_I[T]$

S がラムダ式の時。関数適用は S のコード実行である。

$\mathcal{A}_I[(\textbf{let}\ (x\ (\textbf{call/cc}\ \lambda k.(S\ k\ S_1\ \ldots\ S_n)))\ T)]$
$\longrightarrow$ (PUSH-L l)
$\mathcal{A}_I[S_1]\ \ldots\ \mathcal{A}_I[S_n]$
$\mathcal{A}_I[S]$
$l:$ (bind x)
$\mathcal{A}_I[T]$

c: 定数の値をスタックに積む。

$\mathcal{A}_I[c] \longrightarrow$ (PUSH-C c)

x: PUSH-V で変数の値をスタックに積む。

$\mathcal{A}_I[x] \longrightarrow$ (PUSH-V (depth x))

$\lambda k x_1 \ldots x_n.T$: スタックに積まれているラベルと値を k と $x_1, \ldots, x_n$ に関係付けて本体を実行する。

$\mathcal{A}_C[\lambda k x_1 \ldots x_n.T] \longrightarrow$ (initialize k $(x_1\ \ldots\ x_n)$)
$\mathcal{A}_I[T]$

図 9：繰返し的 Scheme プログラムからのコード生成規則 $\mathcal{A}_I$

うになる。

$$
\begin{aligned}
&(\mathbf{define}\ f\ \ \lambda x_1 \ldots x_n.M)\\
&\xrightarrow{\text{IS}} (\mathbf{define}\ f\ \ \lambda k x_1 \ldots x_n.\mathrm{I}_k[M])\\
&\xrightarrow{\mathcal{A}_I} f:\ (\text{initialize}\ k\ (x_1\ \ldots\ x_n))\\
&\qquad\qquad \mathcal{A}_I[\mathrm{I}_k[M]]
\end{aligned}
$$

$$
\begin{aligned}
&(\mathbf{define}\ f\ \ \lambda x_1 \ldots x_n.M)\\
&\xrightarrow{\text{CPS}} (\mathbf{define}\ f\ \ \lambda k x_1 \ldots x_n.\mathcal{F}[\![M]\!]k)\\
&\xrightarrow{\mathcal{A}_C} f:\ (\text{initialize}\ k\ (x_1\ \ldots\ x_n))\\
&\qquad\qquad \mathcal{A}_C[\mathcal{F}[\![M]\!]k]
\end{aligned}
$$

上記 2 式より、Core Scheme の任意の式 M に対して $\mathcal{A}_I[\mathrm{I}_k[M]] \equiv \mathcal{A}_C[\mathcal{F}[\![M]\!]k]$ を証明すればよい。ただし、$\mathcal{A}_C[\mathcal{F}[\![M]\!]k]$ は式 M の CPS 変換後の式 $\mathcal{F}[\![M]\!]k$ をβ簡約してからコードを生成することを意味する。

この証明の準備として、繰返し的 Scheme プログラムの式を CPS 変換する規則を図 11 に定義する。変換 C_k は、

1. let 文で行われる基本演算子適用に、let 文の変数束縛と本体から変形されるラムダ式をコンティニュエーションとして渡し、
2. call/cc によって捉えられるコンティニュエーションを示す変数を、コンティニュエーションを表わすラムダ式で置換する

ことによって CPS 変換を行う。

$\mathcal{A}_I[\mathrm{I}_k[M]] \equiv \mathcal{A}_C[\mathcal{F}[\![M]\!]k]$ を証明するには、任意の Core Scheme の式 M に対して次を証明すればよい。

$$
\begin{aligned}
\mathcal{A}_I[\mathrm{I}_k[M]] &\equiv \mathcal{A}_C[\mathrm{C}_k[\mathrm{I}_k[M]] \quad (1)\\
\mathcal{A}_C[\mathcal{F}[\![M]\!]k] &\equiv \mathcal{A}_C[\mathrm{C}_k[\mathrm{I}_k[M]]] \quad (2)
\end{aligned}
$$

これらは Core Scheme の任意の式に対して以下のことを示している。

(1) : 繰返し的 Scheme プログラムに変換して生成したコードと、繰返し的 Scheme プログラムを経由した CPS プログラムから生成したコードが等しいこと

(2) : 直接 CPS 変換して生成したコードと、繰返し的 Scheme プログラムを経由した CPS プログラムから生成したコードが等しいこと

これらの変換関係は図 10 のダイアグラムに示した。

(1)、(2) はともにプログラムの大きさに関する帰納法で証明される。(代数的には図 10 のダイアグラムが "commute" することを示すことになる。)

5 まとめ

Scheme のサブセットである Core Scheme プログラムに対して、関数の再帰的な呼出しを **call/cc** 構文を使って末尾再帰風に記述する繰返し的 Scheme プログラムを定義し、Core Scheme の再帰的なプログラムからの変換規則を与えた。繰返し的プログラムからのソースコード生成規則を与え、CPS 法によるコンパイルと同じコードを生成することを示した。

参考文献

1) M. Felleisen, C. Flanagan, A. Sabry, and B. F. Duba. *The Essence of Compiling with Continuation.* ACM SIGPLAN notices, 28, 6, June 1993.

2) W. Clinger and J. Rees, editors. *Revised[4] Report on the Algorithmic Language Scheme.*

3) 伊藤, 田村, 和田. Lisp コンパイラとインタプリタの処理速度の理論的比較. 情報処理学会記号処理研究会 23-3(1983).

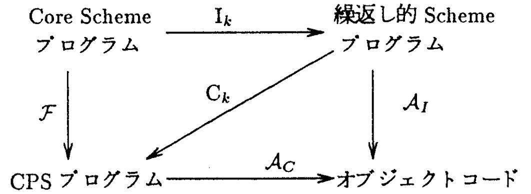

図 10 : 2 つのコンパイル法を示すダイアグラム

$$
\begin{aligned}
\mathrm{C}_k[(k\ S)] &= (k\ \mathrm{D}[S])\\
\mathrm{C}_k[(\mathbf{if}\ S\ T_1\ T_2)] &= (\mathbf{if}\ \mathrm{D}[S]\ \mathrm{C}_k[T_1]\ \ \mathrm{C}_k[T_2])\\
\mathrm{C}_k[(\mathbf{let}\ (x\ S)\ T)] &= (\mathbf{let}\ (x\ \mathrm{D}[S])\ \mathrm{C}_k[T])\\
\mathrm{C}_k[(\mathbf{let}\ (x\ (O\ S_1\ \ldots\ S_n))\ T)] &= \begin{cases} (O'\ k\ \mathrm{D}[S_1]\ \ldots\ \mathrm{D}[S_n]) & T=(k\ x)\ \text{の場合}\\ (O'\ (\lambda x.\mathrm{C}_k[T])\ \mathrm{D}[S_1]\ \ldots\ \mathrm{D}[S_n]) & \text{その他の場合}\end{cases}\\
\mathrm{C}_k[(\mathbf{let}\ (x\ (\mathbf{call/cc}\ \lambda k'.(S\ k'\ S_1\ \ldots\ S_n)))\ T)] &= (\mathrm{D}[S]\ (\lambda x.\mathrm{C}_k[T])\ \mathrm{D}[S_1]\ \ldots\ \mathrm{D}[S_n])\\
\mathrm{C}_k[(S\ k\ S_1\ \ldots\ S_n)] &= (\mathrm{D}[S]\ k\ \mathrm{D}[S_1]\ \ldots\ \mathrm{D}[S_n])\\
\mathrm{D}[c] &= c\\
\mathrm{D}[x] &= x\\
\mathrm{D}[\lambda k x_1 \ldots x_n.T] &= \lambda k x_1 \ldots x_n.\mathrm{C}_k[T]
\end{aligned}
$$

図 11 : 繰返し的 Scheme プログラムから CPS への変換規則