SECDR-Scheme

林　祥一
s-haya@rst.fujixerox.co.jp

平成 5 年 7 月 6 日

概 要

SECDR-Scheme では、Scheme のプログラムは仮想マシンの機械語にコンパイルされてから、仮想マシンによって実行が行なわれます。SECDR-Scheme の仮想マシンは、その名前からも容易に想像できますように、SECD マシンの変形の一つです。

つまり本処理系は、操作的意味を与えるために考案された抽象機械がそのまま動いているようなものです。そしてその抽象機械のわかり易さを優先したために、実行速度に関しては特別な配慮を行なっていません。ですから速度については、はっきりいって速くはない処理系です。

しかし、Scheme 言語についても充分な学習や実験が行えるように、R4RS[4] の仕様に準拠した処理系になっています。具体的には、R4RS を満たしていれば利用できる jaffer のライブラリ slib, 数式処理システム jacal が問題なく動きます。(slib に含まれるオブジェクト指向システム yasos も利用できます。) emacs 上で利用可能なソースレベルデバッガ psd も SECDR-Scheme 特有のマクロを含まなければ利用できることを確認しています。その他、動作することを確認しているものについては README を参照して下さい。

また、もともと R4RS 対応ではなかったために、多少移植の手間を要したもの等に関しては附属のライブラリとして secdrlib の下に添付してあります。この中には schelog という Scheme で書かれた Prolog や、tiny-clos, meroon, methods といったオブジェクト指向ライブラリ, 疑似並列処理を実現する pcall, モジュール管理のための module 等が含まれています。また secdrlib の下には、独自に作成した非決定性のオペレータ amb のライブラリも添付してあります。これは Henderson の著書による方法とはやや異なり、call/cc によって実現されています。

本稿では、はじめにこの SECDR-Scheme の R4RS に対して拡張されている部分について説明します。次に SECDR-Scheme の仮想マシンである SECD マシンについて簡単に説明します。

付録には、添付した amb ライブラリの説明等の継続に関する内容をまとめてあります。付録 C は、以前に Griffin[3] の日本語訳をしたことがありましたので、その中から関係する部分を抜粋[1] しただけのものです。

目 次

1　マニュアル　2

2　SECDR マシンについて　8
2.1　仮想マシンの概要 8
2.2　SECDR マシンのアーキテクチャ 9
2.3　SECDR マシンの Scheme からの操作 . . 12

A　Scheme における継続の応用　13
A.1　call-with-current-continuation の概要 . . 13
A.2　簡単な応用例 14
A.2.1　block と return-from の定義 . . . 14
A.2.2　catch と throw の定義 14
A.3　コルーチンの実現方法 15
A.4　非決定的基本演算と後戻りプログラム . . 15
A.4.1　Scheme への非決定性の導入 . . . 15
A.4.2　amb と fail の実現法 16
A.4.3　全解収集 17

B　マクロ機能を持たない Scheme のために　18

C　ISWIM から Idealized Scheme へ　18

[1] 抜粋した部分の内容からすれば、本来は Felleisen の文献から引用するべきなのですが。

1 マニュアル

ここでは R4RS[4] に対して拡張されているものだけを説明します。またこの処理系は、#v (void) を持っています。そして特に断らない限り、以下に示す procedure は #v を返します。

() special form

() は値() を持つ

nil variable

nil は値() を持つ

t variable

t は値#t を持つ

[ロードとコンパイル]

(load ⟨filename⟩) procedure
(load ⟨filename⟩ ⟨noisily-flg⟩) procedure
(load ⟨filename⟩ ⟨noisily-flg⟩ ⟨error-flg⟩) procedure

⟨filename⟩ は Scheme プログラムのソースファイル名の string であるべきである。
⟨noisily-flg⟩, ⟨error-flg⟩ は指定されなければ#f が指定された場合と同様に振舞う。
⟨noisily-flg⟩ が#f か() であれば、ロード中の評価結果は何も表示されない。#f か() でないならば、ロードされた式の結果を標準出力に表示する。
⟨error-flg⟩ が#f か() であれば、ロードが成功した場合に返す値は#v である。もしも指定されたファイルが見つからない場合はエラーとなる。
#f か() でないならば、正常にロードが行われた場合には#t を返す。もしも指定されたファイルが見つからない場合には#f を返し、エラーとはならない。

(bin-load ⟨filename⟩) procedure
(bin-load ⟨filename⟩ ⟨noisily-flg⟩) procedure
(bin-load ⟨filename⟩ ⟨noisily-flg⟩⟨error-flg⟩) procedure

⟨filename⟩ は Scheme プログラムのコンパイルされたファイル名の string であるべきである。
⟨noisily-flg⟩, ⟨error-flg⟩ は指定されなければ#f が指定された場合と同様に振舞う。
⟨noisily-flg⟩ が#f か() であれば、ロード中の評価結果は何も表示されない。#f か() でないならば、ロードされた式の結果を標準出力に表示する。
⟨error-flg⟩ が#f か() であれば、ロードが成功した場合に返す値は#v である。もしも指定されたファイルが見つからない場合はエラーとなる。
#f か() でないならば、正常にロードが行われた場合には#t を返す。もしも指定されたファイルが見つからない場合には#f を返し、エラーとはならない。

(ex-load ⟨filename⟩) procedure
(ex-load ⟨filename⟩ ⟨noisily-flg⟩) procedure
(ex-load ⟨filename⟩ ⟨noisily-flg⟩ ⟨error-flg⟩) procedure

⟨filename⟩ は Scheme プログラムのソースまたはコンパイルされたファイル名の string であるべきである。
⟨noisily-flg⟩, ⟨error-flg⟩ は指定されなければ#f が指定された場合と同様に振舞う。
⟨noisily-flg⟩ が#f か() であれば、ロード中の評価結果は何も表示されない。#f か() でないならば、ロードされた式の結果を標準出力に表示する。
⟨error-flg⟩ が#f か() であれば、ロードが成功した場合に返す値は#v である。もしも指定されたファイルが見つからない場合はエラーとなる。#f か() でないならば、正常にロードが行われた場合には#t を返す。もしも指定されたファイルが見つからない場合には #f を返し、エラーとはならない。
⟨filename⟩ の string の最後が ".scm" であればソースであると解釈し、".bin" であればコンパイルされたものであると解釈する。それ以外の場合はソースであるものと判断し、ロードを行うが、もしもそのファイルが見つからなかった場合には、拡張子が省略されたものと判断し、⟨filename⟩ の string の最後に".bin" あるいは ".scm"を補ったファイルをロードする。
両方存在する場合は、常に ".bin" が優先して選択される。従って ".bin" 拡張子を持たないコンパイル済みのファイルをロードする場合は、必ず bin-load を用いなければならない。

-*-compile-*- variable

トップレベルの READ-EVAL-PRINT(実際には READ-COMPILE-EXEC-PRINT) ループなどで使

われる標準のコンパイラ手続きを設定する。初期状態では以下の`compile` が設定されている。

`(compile` ⟨expr⟩`)` procedure
`(compile` ⟨expr⟩ ⟨mode⟩`)` procedure

⟨expr⟩ をコンパイルしたコードを返す。⟨mode⟩ に`()` 以外が指定された場合にはマクロの展開だけを行った結果を返す。

`(macro-expand` ⟨expr⟩`)` procedure

⟨expr⟩ のマクロを展開した結果を返す。
`(compile` ⟨expr⟩ `#t)` に同じである。

`(disassemble` ⟨code⟩`)` procedure

⟨code⟩ をディスアセンブルした結果を表示する。したがって ⟨code⟩ はコンパイルされた SECDR マシンのコードであるべきである。

`(compile-file` ⟨sourcefile1⟩ ... ⟨outfile⟩`)` procedure

⟨sourcefile1⟩ ... は、全て既に存在する Scheme プログラムのソースファイル名の string であるべきである。これらのファイルをコンパイルした結果を ⟨outfile⟩ に string で指定されたファイルに出力する。

`(%current-dump%)` procedure

呼び出された時点での D レジスタ (DUMP) の値を返す。
(通常、ユーザはこの procedure を呼び出すべきではない。)

[エバリュータ]

`-*-eval-hook-*-` variable
`-*-exec-hook-*-` variable

各々トップレベルの READ-EVAL-PRINT ループなどで使われる標準の評価, 実行手続きを設定する。初期状態では以下の`eval`, `exec` が設定されている。例えばマクロの展開機能を拡張した`macro:eval` を自分で定義した場合などは、それを設定することによって、トップレベルにおいても`macro:eval` の方を用いることができる。

`(eval` ⟨expr⟩`)` procedure

⟨expr⟩ をトップレベルの環境下で評価し、その値を返す。`(exec (compile` ⟨expr⟩`))` に等価である。

`(exec` ⟨compiled-expr⟩`)` procedure

⟨compiled-expr⟩ をトップレベルの環境下で実行し、その値を返す。

[エラー]

`-*-error-hook-*-` variable

エラーやインタラプトが発生した時にこの値の手続きが起動される。

`(error` ⟨msg⟩ ⟨irritant1⟩ ...`)` procedure

⟨msg⟩ は string であるべきである。エラーであることと、⟨msg⟩,⟨irritant⟩ を表示し、`-*-error-hook-*-` に設定されている手続きを起動する[2] 。

[マシン操作]

`(%master->list%)` procedure

Master の SECDR マシンの 5 つのレジスタの内容をリストにして返す。

`(%list->master%` ⟨list⟩`)` procedure

⟨list⟩ の内容を Master の SECDR マシンの各レジスタに設定する。従って ⟨list⟩ は長さ 5 のリストであるべきである。

`(%reset-master%)` procedure

Slave のマシンがこれを評価すると、Master のマシンの全てのレジスタをクリアする。すなわち `(%list->master% '(()()()()()))` と同等の結果をもたらす。ただし、Master のマシンがこれを評価した場合は何も行うことなくこれを無視する。

[マクロ]

[2] この処理系では仮想的なマシンが 2 台 (Master, Slave) 用意されている。通常動いている Master のマシンはこの呼び出し時点で停止し、上述の処理は Slave が代わって行う。

(macro ⟨keyword⟩ ⟨transformer⟩)　macro special form

マクロスペシャルフォームのキーワードとして⟨keyword⟩を定義する。⟨transformer⟩は、マクロスペシャルフォーム全体を受け取って、それを展開し、置き換えるべき式を返す一引数の手続きであるべきである。

extend-syntax　macro special form

extend.example ファイルを参照のこと。

⟨ 参考 ⟩ jaffer の slib に含まれるマクロを用いたもの (fluidlet.scm, yasos.scm, collect.scm) では、マクロは決まった使われ方しかされていないので、このextend-syntax を用いて、次の要領で書き換えればSECDR-Scheme のマクロで実行できる。その決まった使われ方とは、

```
(define-syntax INSTANCE-DISPATCHER
  (syntax-rules ()
    ((instance-dispatcher inst)
     (cdr inst))))
```

といったように、

```
(define-syntax ⟨NAME⟩
  (syntax-rules () ⟨BODY⟩))
```

の形態をしているので、これを

```
(extend-syntax (⟨NAME⟩)
  ⟨BODY⟩)
```

のように書き換える。

例えば上の例は

```
(extend-syntax (INSTANCE-DISPATCHER)
  ((instance-dispatcher inst)
   (cdr inst)))
```

とすれば良い。

syntax-match?　procedure

extend-syntax が用いる procedure である。

[大域変数定義]

global-define　special form

define と同様であるが、常にトップレベルで評価されたと同じ効果を持つ。

[ガーベジコレクション]

(gc)　procedure

強制的にガーベジコレクションを実行させる。

(gc-verbose-off)　procedure
(gc-verbose-on)　procedure

ガーベジコレクションの実行を標準出力に表示するか否かの指定をする。

[新しいシンボル作成]

(gensym)　procedure
(make-temp-symbol)　procedure
(gensym ⟨string⟩)　procedure
(make-temp-symbol ⟨string⟩)　procedure

新しいシンボルを生成し、それを返す。⟨string⟩の指定の有る無しによって次のようになる。

```
(gensym)-->     G123
(gensym "temp") -->     temp123
```

[属性リスト]

(get ⟨symbol⟩ ⟨attribute⟩)　procedure
(put ⟨symbol⟩ ⟨attribute⟩ ⟨value⟩)　procedure

⟨symbol⟩ の属性リストに対する操作を行なう。put は#v を返し、get は指定された属性値を返す。

[システム関係]

(quit)　procedure

SECDR-Scheme の実行を終了する。

(new-segment ⟨n⟩)　procedure

ヒープ領域を広げる。⟨n⟩ は広げたいセグメントの数で、正の整数であるべきである。（１セグメントの大きさはシステムによる。）実際に拡張できたセグメント数をメッセージ出力する。`#v` を返す。

`(system` ⟨string⟩`)` procedure

⟨string⟩ の内容をホスト環境に渡し、コマンドプロセッサに実行させる。その結果の値を返す。

`(pwd)` procedure

カレントディレクトリのフルパス名を string として返す。

`(cd)` procedure
`(cd` ⟨pathname⟩`)` procedure

⟨pathname⟩ を省略すると、カレントディレクトリをホームディレクトリに設定する。⟨pathname⟩ は string であるべきである。指定されたパスへカレントディレクトリを変更する。成功すると`#t`、失敗ならば`#f` を返す。

`(chdir` ⟨pathname⟩`)` procedure

⟨pathname⟩ は string であるべきである。指定されたパスへカレントディレクトリを変更する。成功すると`#t`、失敗ならば`#f` を返す。

`(getenv` ⟨name⟩`)` procedure

⟨name⟩ は string であるべきである。名前が ⟨name⟩ に等しいシステムの環境変数の値を string として返す。もしもそのような環境変数が定義されていない場合は`#f` を返す。

`(software-type)` procedure

実行している環境が unix 上であれば`unix`, Macintosh 上であれば`macos`, MS-DOS 上であれば`msdos` を返す。

`(ls)` procedure
`(ls` ⟨string⟩`)` procedure

shell に ls ⟨string⟩ を実行させる。

`(rm` ⟨filename⟩`)` procedure

shell に rm ⟨filename⟩ を実行させる。

`(ed)` procedure
`(ed` ⟨filename⟩`)` procedure

エディタを起動する。

`my-ls` variable
`my-rm` variable
`my-editor` variable

ls, rm, ed で実行したいシステムのコマンド名（＋オプション）を文字列としてこれらの変数に定義する。再定義可能。

[TIME]

`(get-internal-time-units)` procedure

`CLOCK_PER_SEC` の値を integer として返す。

`(get-internal-run-time)` procedure

実行時間の CLOCK 単位の値を integer として返す。

[トレース]

`(trace` ⟨sym1⟩ ⟨sym2⟩ ...`)` procedure
`(untrace` ⟨sym1⟩ ⟨sym2⟩ ...`)` procedure

⟨sym1⟩, ⟨sym2⟩, ... に対しトレースの設定、あるいは解除を行なう。

`set-trace` procedure

`trace` が用いる procedure である。

[制御]

`(sequence` ⟨expr⟩`)` macro special form

以下のように定義されている。

```
(macro sequence (lambda (expr)
  `((lambda () ,@(cdr expr))) ))
```

注意するべきことは、`begin` とは異なり、⟨expr⟩ 内の define 文を特別扱いしないということである。

`while` macro special form

`when` macro special form

`unless` macro special form

各々以下のように定義されている。

```
(macro while (lambda (while-macro)
  (apply
   (lambda (pred . body)
     (let ((while-loop (gensym))
           (while-res (gensym)))
       `(letrec ((,while-loop
           (lambda (,while-res)
             (if ,pred
                 (,while-loop
                  (begin ,@body))
                 ,while-res))))
          (,while-loop #f))))
   (cdr while-macro))))

(macro when
  (lambda (args)
    `(if ,(cadr args)
         (begin ,@(cddr args))
         #f)))

(macro unless
  (lambda (args)
    `(if ,(cadr args)
         #t
         (begin ,@(cddr args)))))
```

`(rec` ⟨var⟩ ⟨expr⟩`)` macro special form

`(letrec ((`⟨var⟩ ⟨expr⟩`))` ⟨var⟩`)` に等価である。`rec` は以下の例に示すように、self-recursive procedure の定義に便利である。

```
((rec loop
  (lambda (n l)
    (if (zero? n)
        l
        (loop (- n 1)(cons n l)))))
 6 '())          --> (1 2 3 4 5 6)
```

以下のように定義されている。

```
(macro rec (lambda (rec-macro)
  (apply
   (lambda (proc body)
     `(letrec ((,proc ,body)) ,proc))
   (cdr rec-macro))))
```

`andmap` procedure

以下のように定義されている。

```
(define (andmap p . args)
  ;; use "first-finish" rule
  (let andmap ((args args) (value #t))
    (if
     (let any-at-end? ((ls args))
       (and (pair? ls)
            (or (not (pair? (car ls)))
                (any-at-end? (cdr ls)))))
     value
     (let ((value
              (apply p (map car args))))
       (and value
            (andmap (map cdr args)
                    value))))))
```

[述語]

`(file-exists?` ⟨file-name⟩`)` procedure

⟨file-name⟩ は string であるべきである。⟨file-name⟩ に指定された名前に一致するファイルが既に存在している場合は`#t` を、そうでなければ`#f` を返す。

`every?` procedure

`some?` procedure

各々以下のように定義されている。

```
(define (every? proc plist)
  (cond
   ((null? plist) #t)
   ((proc (car plist))
    (every? proc (cdr plist)))
   (else #f)))

(define (some? proc plist)
```

```
(cond
 ((null? plist) #f)
 ((proc (car plist)) #t)
 (else (some? proc (cdr plist)))))
```

(atom? ⟨obj⟩) procedure
(closure? ⟨obj⟩) procedure
(continuation? ⟨obj⟩) procedure
(primitive-procedure? ⟨obj⟩) procedure
(proper-rational? ⟨obj⟩) procedure
(proper-real? ⟨obj⟩) procedure
(promise? ⟨obj⟩) procedure

⟨obj⟩ が各々の type であるとき#t、そうでなければ #f を返す。

(set-promise ⟨obj⟩) procedure

(promise? ⟨obj⟩) が#t を返すようにする。delay で用いている。

[型変換]

char->string procedure

以下のように定義されている。

```
(define (char->string c)
  (string-set! "@" 0 c))
```

[ボックス型]

(box) procedure
(box ⟨object⟩) procedure

⟨object⟩ を持つ box を返す。ボックス型の差したてマクロ文字が標準で用意されている。それは#&で、(box ⟨object⟩) は#&(⟨object⟩) と記述しても良い。

(unbox ⟨box-object⟩) procedure

⟨box-object⟩ は box でなければならない。⟨box-object⟩ の中の object を取り出し、それを返す。

(box? ⟨object⟩) procedure

⟨object⟩ が box であるときは#t、そうでないときは #f を返す。

(set-box! ⟨box-object⟩ ⟨object⟩) procedure

⟨box-object⟩ は box でなければならない。⟨box-object⟩ の中を ⟨object⟩ で置き換える。

[ストリーム]

cons-stream macro special form
head procedure
tail procedure

各々以下のように定義されている。

```
(macro cons-stream (lambda (l)
  `(cons ,(cadr l) (delay ,(caddr l)))))

(define (head stream)(car stream))

(define (tail stream)(force (cdr stream)))
```

[入出力]

(spaces ⟨num⟩ ⟨outport⟩) procedure

⟨num⟩ で指定された数のスペースを ⟨outport⟩ に出力する。

(print-list ⟨mode⟩ ⟨list⟩ ⟨outport⟩) procedure

⟨list⟩ を ⟨outport⟩ に出力する。⟨mode⟩ に 0 が指定されると、⟨list⟩ の各要素はdisplay で出力されたようになり、その他の値が指定されると、各要素はwrite で出力されたようになる。

pretty-print procedure

簡易版 pretty-print

prompt-read procedure

以下のように定義されている。

```
(define (prompt-read prompt)
  (display prompt) (read))
```

printf procedure

C 言語の printf に似たもの

`%format%` procedure

簡易版 format

`(uniq-print` ⟨term⟩`)` procedure

⟨term⟩ に循環 list または vector が含まれる場合は、それにラベルをつけて pretty-print 形式で表示する。

`(uniq-write` ⟨term⟩`)` procedure

⟨term⟩ に循環 list または vector が含まれる場合は、それにラベルをつけて表示する。

`(uniq-read)` procedure

循環 list または vector のラベルによる表現形式を正しく読みとる。

`(u-conv` ⟨term⟩`)` procedure

⟨term⟩ に循環 list または vector が含まれる場合は、⟨term⟩ をラベルによる表現に破壊的に変換し、その値を返す。
(注: 元の値に戻すためにはこの返す値を必ずとっておく必要がある。すなわち返す値と ⟨term⟩ は異なる。)

`(uu-conv` ⟨term⟩`)` procedure

循環 list または vector のラベルによる表現形式を含む ⟨term⟩ を元の状態に破壊的に戻し、その値を返す。
(注: ⟨term⟩ 自身を元の値に戻すためには、この返す値を必ず設定し直す必要がある。)

注) 蛇足ながら`(uu-conv (u-conv` ⟨term⟩`))` はもちろん正しく ⟨term⟩ を元に戻す。

[R2RS]

`-1+` procedure
`1+` procedure

[その他]

`simplest-rational` procedure

`rationalize` が用いる procedure である。

`(list-width` ⟨obj⟩`)` procedure

⟨obj⟩ の印字形式が何文字 (byte) になるかを返す。

`(qsort` ⟨seq⟩ ⟨pre⟩`)` procedure

⟨pre⟩ は比較のための述語、⟨seq⟩ は list または vector であるべきである。クイックソートによって並べ換えた値を返す。

`(qsort!` ⟨seq⟩ ⟨pre⟩`)` procedure

⟨pre⟩ は比較のための述語、⟨seq⟩ は vector であるべきである。クイックソートによって ⟨seq⟩ で与えられた vector の要素を並べ換える。そしてその値を返す。
(副作用を期待する場合に用いる。)

---

2 SECDR マシンについて

2.1 仮想マシンの概要

本 Scheme の処理系では、Scheme のプログラムは仮想マシンの機械語にコンパイルされてから、仮想マシンによって実行が行なわれます。ここでは主に、この仮想マシンについて述べることにします。

一般に関数型言語の実現モデルは、環境スタックモデル, データフローモデル, リダクションモデル等に分類することができます。

環境スタックモデルは、変数の結合情報と実行順序の情報を”環境”として実現し、関数の実行順序に従って環境をスタックすることにより、関数型言語の実行メカニズムを実現する計算モデルです。そのよく知られた代表的な例の一つとして SECD マシンがあります。

そして SECDR-Scheme の仮想マシンは、その名前からも容易に想像できますように、この SECD マシンの変形の一つです。

環境スタックマシンはハードウェアスタックや関数型言語向きのスタック操作命令などを用意することにより関数型言語の実行を高速化する計算機です。この環境スタックマシンでは、スタックを用いることにより、変数の結合環境や実行情報を効率よく管理することができます。しかし、関数の並列実行や、遅延評価、高階関数機能の実現に対しては必ずしも効率的なものではありません。

環境スタックマシンの上述の欠点に対して有効なものとして、例えばLazyML等のインプリメントに用いられているG-マシンがあります。G-マシンはリダクションモデルの一つであり、SGCDマシンと呼ばれることもあります。つまりSECDマシンのE(環境)の代わりにG(グラフ)を持ち、グラフリダクションを行なうマシンです。

さて、ここではSECDR-Schemeに直接関係する環境スタックモデルのSECD(SECDR)マシンについてのみ簡単に解説することにしますので、グラフリダクションやその実装等にも興味をお持ちになったら、例えば[5]等をお読みになって下さい。

最初にSECDマシンの歴史を簡単に振り返っておきましょう。

Peter Landinは彼の初期の論文で、λ算法と機械および高水準言語(特にALGOL60)との関係を論じています。そしてSECDと呼ぶ抽象機械によって式の評価をどの様に機械的に行うのかを述べています。1966年にLandinは一つの言語(の一族)Iswim(If You See What I Meanの略)を導入し、数多くの重要な構文上の考え方、意味上の考え方を示しました。つまりSECD機械は、Landinが意味論上の革新に対する功績の中で、関数プログラムを実行するための単純な抽象機械として提示したものです。

その後Peter HendersonがLispkit Lisp(中置演算子と代数的な構文を持った純正Lispの一つ)のインプリメントのために、やや変形したものですが、このSECDマシンを用いています。Hendersonの著書[6]には、SECDマシンの解説と実装法が詳しく解説されています。従ってSECDマシンについて詳細を知りたい方は、この著書を読まれると良いでしょう。

そして、このSECDR-Schemeの仮想マシンであるSECDRマシンは、このHendersonの示したSECDマシンに基づいて作成されたものです。しかし、主に末尾再帰に対処するために、マシンのいくつかの状態遷移は異なったものとなっています。つまり、さらに変形したものの一つとなっているということです。

ようするに、そもそもこのSECDRマシンの最初の原型であるSECDマシンは、Landinによって関数型言語Iswimやそれらの形式的な操作的意味論のために考案された抽象機械であったものです。そしてSECDR-Schemeの処理系は、操作的意味を与えるこの抽象機械がそのまま動いているようなものです。(Schemeの表示的意味は、その一つの仕様書であるR4RSにおいて形式的に与えられています。)

2.2 SECDRマシンのアーキテクチャ

SECDRマシンは、以下のように5つのレジスタを持つ環境スタックマシンです。

S: stack

式の値を計算する時の中間結果を入れる。

E: environment

計算の途中で各変数に束縛される値を入れる。

C: control list

実行されるべき機械語プログラムを入れる。

D: dump

新しい関数呼び出しが起こった時に他のレジスタの内容をしまっておく。スタックの一種。

R: return

戻り値を格納する。

SECDRマシンの19個の命令と各々の状態遷移を、以下の表に示します。説明不足で理解できない部分があるかも知れませんが、その場合は上述の[6]を参考にして下さい。

LD: #0

```
S E (LD (i . j) . C) D R ==> S E C D r
 where r = (list-ref (list-ref E i) j)
```

TLD: #1

```
S E (TLD (i . j) . C) D R ==> S E C D r
 where r = (list-tail (list-ref E i) j)
```

GLD: #2

```
S E (GLD sym . C) D R ==> S E C D r
 where r = gloabl value of sym
```

LDC: #3

```
S E (LDC const . C) D R ==> S E C D const
```

LDF: #4

```
S E (LDF code . C) D R
      ==> S E C D (closure of code and E)
```

AP: #5

```
(args . S) E (AP . C) D op ==>
```

case: op is closure

```
() (args . env) code (S E C . D) NIL
where code is closure code of op and
env is closure environment of op.
```

case: op is primivite procedure

```
First, execute primitive procedure op
with arguments args and set registers
to S E C D r, where r is the return
value of primitine procedure op.
```

case: op is continuation

```
s e c d r
where r is first element of args and
s e c d are saved registers in op.
```

TAP: #6

```
(args . S) E (TAP . C) D op ==>
```

case: op is closure

```
() (args . env) code D NIL
where code is closure code of op and
env is closure environment of op.
```

case: op is primivite procedure

```
First, execute primitive procedure op
with arguments args and set registers
to S E C D r, where r is the return
value of primitine procedure op.
```

case: op is continuation

```
s e c d r
where x is the first element of args
and s e c d r are saved registers in
op.
```

PUSH: #7

```
S E (PUSH . C) D R ==> (R . S) E C D R
```

RTN: #8

```
S E (RTN . C) (s e c . d) R ==> s e c d R
```

SEL: #9

```
S E (SEL ct cf . C) D test
           ==> S E cx (S E C . D) test
        where cx = (if test ct cf)
```

TSEL: #10

```
S E (TSEL ct cf . C) D test
           ==> S E cx D test
        where cx = (if test ct cf)
```

ASSIG: #11

```
S E (ASSIG (i . j) . C) D R ==> S E' C D R
 where E' is made by
  (set-car! (list-tail (list-ref E i) j) R)
```

TASSIG: #12

```
S E (TASSIG (i . j) . C) D R ==> S E' C D R
 where E' is made by
   (if (zero? j)
     (set-car! (list-tail E i) R)
     (set-cdr! (list-tail (list-ref E i)
                          (- j 1)) R))
```

GASSIG: #13

```
S E (GASSIG sym . C) D R ==> S E C D R
 where global value of sym = R
```

DEF: #14

```
S E (DEF sym . C) D R ==> S E C D sym
 where global value of sym = R
```

PUSHCONS: #15

```
(s . S) E (PUSHCONS . C) D R
           ==> ((R . s) . S) E C D R
```

SAVE: #16

```
S E (SAVE C1 . C2) D R
           ==> S E C1 (S E C2 . D) R
```

EXEC: #17

```
S E (EXEC . C) D code
        ==> NIL NIL code (S E C . D) NIL
```

STOP: #18

```
S E (STOP . C) D R ==> Stop SECDR execution
```

参考までに以下に Henderson の SECD マシンの状態遷移も示しておきます。

LD: load

```
S E (LD i . C) D --> (x . S) E C D
                     where x=locate(i, e)
```

LDC: load constant

```
S E (LDC x . C) D --> (x . S) E C D
```

LDF: load function

```
S E (LDF C' . C) D --> ((C' . E) . S) E C D
```

AP: apply function

```
((C' . E') v . S) E (AP . C) D
            --> NIL (v . E') C' (S E C . D)
```

RTN: return

```
(x) E' (RTN) (S E C . D) --> (x . S) E C D
```

DUM: create dummy environment

```
S E (DUM . C) D --> S (O . E) C D
```

RAP: recursive apply

```
((C' . E') v . S) (O . E) (RAP . C) D
      --> NIL rplaca(E', v) C' (S E C . D)
```

SEL: select subcontrol

```
(x . S) E (SEL cT cF . C) D
                --> S E Cx (C . D)
```

JOIN: rejoin main control

```
S E (JOIN) (C . D) --> S E C D
```

CAR:

```
((a . b) . S) E (CAR . C) D
                --> (a . S) E C D
```

CDR:

```
((a . b) . S) E (CDR . C) D
                --> (b . S) E C D
```

ATOM:

```
(a . S) E (ATOM . C) D --> (t . S) E C D
```

CONS:

```
(a b . S) E (CONS . C) D
                --> ((a . b) . S) E C D
```

EQ:

```
(a b . S) E (EQ . C) D
                --> (b=a . S) E C D
```

ADD:

```
(a b . S) E (ADD . C) D
                --> (b+a . S) E C D
```

SUB:

```
(a b . S) E (SUB . C) D
                --> (b-a . S) E C D
```

MUL:

```
(a b . S) E (MUL . C) D
                --> (b*a . S) E C D
```

DIV:

```
(a b . S) E (DIV . C) D
                --> (b/a . S) E C D
```

REM:

```
(a b . S) E (REM . C) D
                --> (b rem a . S) E C D
```

LEQ:

```
(a b . S) E (LEQ . C) D
                --> (b=<a . S) E C D
```

STOP:

```
S E (STOP) D --> S E (STOP) D
```

2.3 SECDR マシンの Scheme からの操作

SECDR-Scheme のコンパイラは Scheme 自身で書かれています。またコンパイル結果である中間コードは、テキストファイルになります。ディスアセンブラも含まれていますので、コンパイル結果を容易に理解することができるでしょう。

ところで、直接この中間コードをテキストで入力し、実行することもできます。また MANUAL の方にも記述しましたように、本処理系では仮想マシンは Master と Slave の 2 台が存在します。そして Master のマシンは、エラーやインタラプトが発生した場合には、その状態を保存したまま停止します。そして Slave のマシンが動き出しますので、Slave 側から Master のレジスタを参照したり変更したりすることができます。(Init.scm の error hook の関数を参考にして下さい。) これらの操作は注意深く行なうことが必要ですが、試してみると仮想マシンに何が起こるのか良くわかるはずです。

以下に例を示します。

```
> (define code
    (compile
     '(define (fact n)
        (if (zero? n)
            1
            (* n (fact (1- n)))))))
; Evaluation took 0.049998 Seconds (0 in gc).
code
> code
; Evaluation took 0 Seconds (0 in gc).
(#4 (#3 () #7 #0 (0 . 0) #15 #2 zero? #5 #10
 (#3 1 #8) (#3 () #7 #3 () #7 #3 () #7 #0
  (0 . 0) #15 #2 1- #5 #15 #2 fact #5 #15 #0
  (0 . 0) #15 #2 * #6 #8) #8) #14 fact #8)
> (disassemble code)


************ DisAssemble Code ************

  (
  LDF       (
      LDC       ()
      PUSH
      LD        (0 . 0)
      PUSHCONS
      GLD       zero?
      AP
      TSEL      (
          LDC       1
          RTN
          )
          (
          LDC       ()
          PUSH
          LDC       ()
          PUSH
          LD        (0 . 0)
          PUSHCONS
          GLD       1-
          AP
          PUSHCONS
          GLD       fact
          AP
          PUSHCONS
          LD        (0 . 0)
          PUSHCONS
          GLD       *
          TAP
          RTN
          )
      RTN
      )
  DEF       fact
  RTN
  )
************ END DisAssemble ************

; Evaluation took 0.116662 Seconds (0 in gc).

> (exec code)
; Evaluation took 0.016666 Seconds (0 in gc).
fact
> (fact 5)
; Evaluation took 0 Seconds (0 in gc).
120

> (compile '(lambda (m n)(+ m n)))
; Evaluation took 0.033332 Seconds (0 in gc).
(#4 (#3 () #7 #0 (0 . 1) #15 #0 (0 . 0) #15
  #2 + #6 #8) #8)
> ((lambda (m n)(+ m n)) 3 2)
; Evaluation took 0.016666 Seconds (0 in gc).
5
> ((exec '(#4 (#3 () #7 #0 (0 . 1) #15 #0
           (0 . 0) #15 #2 + #6 #8) #8)) 3 2)
; Evaluation took 0.033332 Seconds (0 in gc).
5
```

A Schemeにおける継続の応用

Scheme を学ぼうとする場合は、例えば Abelson と Sussman による「Structure and Interpretation of Computer Programs」[1] という良書があるので、これを読まれるとよいと思う。しかし、この文献には継続についての解説がないので、以下では継続についてのみ解説することにする。以下に示したプログラムは、SECDR-Scheme での実行を確認している。ただし Scheme の仕様書 [4] では、まだマクロの仕様が正式には規定されていないので、マクロの仕様は処理系によって異なる。実際、SECDR-Scheme のマクロは仕様書 [4] の appendix に記述されているものとは異なる。それゆえ、他の処理系を用いる場合にはマクロの仕様によって若干の手直しが必要となるかもしれないことを断っておく。(付録 B 参照)

A.1 call-with-current-continuation の概要

```
(call-with-current-continuation proc)
```

proc は 1 引数の手続きでなければならない。手続き call-with-current-continuation は現在の継続をエスケープ手続きとして一まとめにし、proc の引数として渡す。エスケープ手続きは 1 引数の Scheme 手続きで、もし後にある値を受け取ると、その後の時点に有効な継続を無視し、そのかわりにエスケープ手続きが作られたときに有効であった継続にその値を渡す。

call-with-current-continuation で作られたエスケープ手続きは、他の Scheme の手続きとまったく同様に無制限の動的存在範囲 (extent) を持つ。それは変数や他のデータ構造に蓄えられてもよいし、望むかぎり多数回呼ぶこともできる。

以下の例は call-with-current-continuation のもっとも普通の使い方を示しているに過ぎない。もし実際のプログラムがこれらの例ほど単純ならば、call-with-current-continuation ほどの能力を持った手続きは必要ない。

例 A.1

```
(call-with-current-continuation
    (lambda (exit)
        (for-each (lambda(x)
            (if (negative? x)
                (exit x)))
          '(54 0 37 -3 245 19))
    #t))                        ==>  -3
```

例 A.2

```
(define list-length (lambda (obj)
  (call-with-current-continuation
    (lambda (return)
      (letrec ((r
          (lambda (obj)
             (cond ((null? obj) 0)
                   ((pair? obj)
                    (+ (r (cdr obj)) 1))
                   (else (return #f))))))
          (r obj))))))


(list-length '(1 2 3 4))     ==>  4
(list-length '(a b . c))     ==>  #f
```

call-with-current-continuation の一般的な使い方は、構造的なループや手続き本体からの非局所的な抜け出しである。しかし、call-with-current-continuation は、もっと広い範囲の高度な制御構造を実現するのに非常に有用である。Scheme 式が評価されるときはいつでも式の結果を要求している継続 (continuation) が存在する。継続は計算の総ての (デフォルトの) 未来を表している。例えば式がトップレベルで評価されるならば、その継続は結果を受け取り, それをスクリーンに印字し, 次の入力を待ち, さらにそれを評価することを永久に続けるであろう。多くの場合、継続はユーザのコードで指定された動きを含んでいる。通常の場合、これらの偏在する継続は背景に隠れていて、プログラマはそれらのことをあまり考えなくてもよい。しかしながら、まれにプログラマは継続を明示的に取り扱う必要があるかもしれない。 call-with-current-continuation は現在の継続と同じように働く手続きを作ることによって、この操作をプログラマに許す。

多くのプログラミング言語は exit,return, または goto という一つ以上の特殊目的のエスケープ機構を取り入れている。しかしながら、1965 年に Peter Landin は J-operater という汎用のエスケープ操作を発明した。John Reynolds はより単純で同程度に強力な機構を 1972 年に示した。1975 年の Sussman と Steele による Scheme の論文に述べられている catch 機構は Reynolds の機構と全く同じである。数人の Scheme のインプリメンタは catch 機構の全能力は特殊構文の機構の代わりに手続きによって備えることができることを紹介し、名前 call-with-current-continuation が 1982 年に作られた。この名前は説明的で

あるが、そのように長い名前の効用についての見解は別れており、ある人々は名前 call/cc を使用する。

例 A.3 (階乗を計算するプログラム)

```
(define (fact n)
  (let ((ans 1)
        (pitch nil))
    (let ((w (call/cc
                (lambda (-c-)
                    (set! pitch -c-)
                    n))))
        (cond ((= w 0) ans)
              (else (set! ans (* ans w))
                    (pitch (-1+ w)))))))
```

よく用いる技法として、current-continuation を以下のように定義しておけば、その時点での継続を取り出すことができる。

```
(define identity (lambda (x) x))


(define (current-continuation)
    (call-with-current-continuation
    identity))
```

このcurrent-continuation を用いると、fact は次のように定義することができる。

例 A.4

```
(define (fact n)
  (let ((ans 1))
    (let ((pitch (current-continuation)))
        (cond ((= n 0) ans)
              (else (set! ans (* ans n))
                    (set! n (-1+ n))
                    (pitch pitch))))))
```

ここで(pitch pitch) は、自分の継続を再び自分自身に束縛している。

A.2 簡単な応用例

ここでは Common Lisp のblock, return-from, catch, throw をcall-with-current-continuation を用いて定義してみることにする。

A.2.1 block と return-from の定義

```
(macro block (lambda (l)
    (let ((name (cadr l))
          (forms (cddr l)))
        `(call-with-current-continuation
            (lambda (,name) ,@forms)))))


(macro return-from (lambda (l)
    (let ((name (cadr l))
          (value (caddr l)))
        `(,name ,value))))
```

A.2.2 catch と throw の定義

```
(define (make-stack)
    (define s '())
    (define (push x)
        (set! s (cons x s)))
    (define (pop)
        (if (null? s)
            (error "Empty stack -- POP")
            (let ((top (car s)))
                (set! s (cdr s))
                top)))
    (define (empty?)
        (null? s))
    (define (print)
        (write s))
    (define (stack->list)
        s)
    (define (initialize)
        (set! s '()))
    (define (dispatch message)
        (cond ((eq? message 'push) push)
              ((eq? message 'pop) (pop))
              ((eq? message 'empty?)
               (empty?))
              ((eq? message 'print)
               (print))
              ((eq? message 'stack->list)
               (stack->list))
              ((eq? message 'initialize)
               (initialize))
              (else
               (error
                "Unknown request -- STACK"
                 message))))
    dispatch)

(define %catchers% (make-stack))

(macro catch (lambda (l)
    (let ((tag (cadr l))(forms (cddr l)))
      `(call-with-current-continuation
          (lambda (-c-)
            ((%catchers% 'push)
             (cons ,tag -c-))
            (let ((ans (begin ,@forms)))
               (%catchers% 'pop)
               ans))))))
```

```
(define (throw tag value)
    (let ((continuation
             (search-catcher tag)))
        (continuation value)))

(define (search-catcher tag)
    (if (%catchers% 'empty?)
        (error "Tag name not found:" tag)
        (let ((ans (%catchers% 'pop)))
            (if (eq? (car ans) tag)
                (cdr ans)
                (search-catcher tag)))))
```

A.3 コルーチンの実現方法

call-with-current-continuation の機構を用いることによって、コルーチンを簡単に実現することができる。次に示す例はその実現方法を示唆するものである。

例 A.5

```
(define (print x)
    (display x)
    (newline))

(define (bar)
    (let ((qwe nil))
        (print 'init)
        (call/cc
          (lambda (-c-) (set! qwe -c-)))
          (print 'bar)
          qwe))

(define (foo)
    (let ((qwe nil))
        (set! qwe (bar))
        (print 'foo)
        (qwe nil)))
```

ここで `(foo)` を実行すると

```
init
bar
foo
bar
foo
...
```

となる。

A.4 非決定的基本演算と後戻りプログラム

McCarthy は次のような性質を持つ 2 項の非決定性の演算子を定義した。[McCarthy1963]

$$amb(e_1, \perp) = e_1$$
$$amb(\perp, e_2) = e_2$$
$$amb(e_1, e_2) = \text{非決定的に } e_1 \text{か } e_2 \text{を選ぶ}$$

式 $amb(e_1, e_2)$ を操作的に読むと、e_1 と e_2 を並列に評価して、早く返された方の値をこの式の値とするということである。

SECDR-Scheme の附属のライブラリには、この amb の一つの変形が含まれている。以下はそのライブラリに関する説明である。

A.4.1 Scheme への非決定性の導入

次の関数は、長さが n であるリスト a の n+1 箇所ある任意の位置に要素を差し込む関数である。

```
(define (insert x a)
    (if (null? a)
        (cons x ())
        (amb (cons x a)
             (cons (car a)
                   (insert x (cdr a))))))
```

ここで、関数`amb` は`(amb exp1 exp2)` を評価すると、`exp1` か`exp2` の何れか一方を返す。`exp1` と`exp2` のどちらかではあるが、2 つの値の何れであるかは定まらないものとする[3] 。

次に示す関数`perm` は任意に並べ変えたリスト (任意の順列) を、また`choice` は 1 から n までの整数の中から任意の値を返す。

```
(define (perm b)
    (if (null? b)
        ()
        (insert (car b) (perm (cdr b)))))

(define (choice n)
    (if (= n 1)
        1
        (amb n (choice (-1+ n)))))
```

次に`(fail)` という式を導入する。プログラム (式) の評価の途中で、`(fail)` という部分式を評価せずに済むような選択の系列があるならば、その選択の系列が実行されるものとする。`amb` と`fail` の意味は、次のように考えるとわかりやすい。まず`amb` と`fail` を含むプログラムを実行する抽象機械があるとする。この機械が`(amb exp1 exp2)`

[3] `exp1` と`exp2` をそれぞれ計算し、次に硬貨を投げてその何れかを選択すると考えればよい。そうすると`(- (amb 1 2) (amb 1 2))` は硬貨を 2 度投げたことになるので、0 となるとは限らないことに注意を要する。

という式の値の計算に出会うと、その機械は自分と全く同じ状態の機械のコピーを一つ作りだし、片方の機械にはexp1 に出会ったのと同じように、他方の機械にはexp2 に出会ったのと同じように計算を進めさせる。そしてどの機械もfail に出会うとそこでその機械は捨てられてなくなると考える。amb とfail がどのように使われるかを 8-QUEEN のパズルを例にして説明する。

```
(define (attacks i j place)
    (if (null? place)
        #f
        (let ((ii (caar place))
              (jj (cdar place)))
            (cond ((= i ii) #t)
                  ((= j jj) #t)
                  ((= (+ i j)
                      (+ ii jj)) #t)
                  ((= (- i j)
                      (- ii jj)) #t)
                  (else
                    (attacks i j
                      (cdr place)))))))

(define (addqueen i n place)
    (let ((j (choice n)))
        (if (attacks i j place)
            (fail)
            (let ((newplace
                   (cons (cons i j)
                         place)))
                (if (= i n)
                    newplace
                    (addqueen (1+ i) n
                        newplace))))))

(define (queen n)
    (addqueen 1 n ()))
```

ここで(attacks i j place) は、女王の配置されているます目のリストplace から、チェス盤のi 行j 列のます目に対し、この配置の中のどれかの女王が利いているときに限って真となる述語である。

非決定性が何箇所かで導入されるような複雑な問題に対しては、非決定的な基本演算を用いて直接定式化する方が後戻りを陽にプログラミングするよりは確実に有利である。その有利さは、最も具合の良い選択がなされるものとして、勝手な選択をすると考えて良い点に由来する。

A.4.2 amb と fail の実現法

これまでamb については、その意味を抽象機械が(amb exp1 exp2) という式の値の計算に出会うと、自分と全く同じ状態の機械のコピーを一つ作り出すとしてきた。その場合、この二つの抽象的な機械は独立に並列に動作しても良いことになる。また、実際に並列に動作する機械が存在しなければ、ある特定の順序 (スケジューリング) に従って逐次的にシミュレートすれば良い。ここではそのスケジューリングをexp1,exp2 の順に行い、また複数の非決定性が導入されている場合は、最も近く (最近) の選択をやり直すという規則とする。これは探索問題の解法として、横型探索ではなく縦型探索をする事に相当する。このようなスケジューリングをする事による欠点は、exp2 を選択すれば計算が停止するが、exp1 を選んだ場合に計算が終了しないといった場合に、全体の計算が停止しないといった問題が起こり得ることである。

さて、後に示すリストによるamb の実現方法では、choice は以下のように展開される。

```
(define (expand-choice n)
  (if (= n 1)
      1
      (call-with-current-continuation
        (lambda (-c-)
          ((%amb-catchers-stack% 'push)
           (cons -c-
             (delay
               (expand-choice (-1+ n)))))
          n))))
```

もしも、choice を以下のように定義したとすると、全ての可能な継続点を計算した後に実行に入ることになる。

```
(define (choice n)
    (if (= n 1)
        1
        (amb (choice (-1+ n)) n)))
```

この場合に気をつけなければならないことは、可能な場合が有限でなければならないということである。例えば任意の 0 以上の整数を返す関数を考えてみることにする。このような無限に可能な継続点が存在する場合には、以下に示すany-numbe1 の定義であれば他の部分の計算が進むが、any-number2 の定義を用いると全ての可能な環境を計算し続けるだけで、他の部分の計算は永久に遅らされる。これは Prolog における左再帰の問題とよく似ている。

```
(define (any-number1)
    (amb 0 (1+ (any-number1))))
(define (any-number2)
    (amb (1+ (any-number2)) 0))
```

以上述べてきたような制約はあるものの、amb とfail はcall-with-current-continuation を用いて、以下のように定義することができる。

```
(define %amb-catchers-stack% (make-stack))

(macro amb (lambda (l)
  (let ((exp1 (cadr l))(exp2 (caddr l)))
    `(call-with-current-continuation
       (lambda (-c-)
          ((%amb-catchers-stack% 'push)
             (cons -c- (delay ,exp2)))
          ,exp1)))))

(define (fail)
  (if (%amb-catchers-stack% 'empty?)
    (error "FAIL evaluated.")
    (let ((first-continuation
           (%amb-catchers-stack% 'pop)))
      ((car first-continuation)
       (force
         (cdr first-continuation))))))
```

以上で、amb とfail をシミュレートすることはできたわけであるが、次のような問題が残る。amb によってコピーされた機械は、いったい何時までとっておけば良いのかということである。一つの機械がfail によって捨てられてなくなってしまった場合、もう一つの機械が動作を始めなければならない。しかし、fail にいつどこで出会うのかは明確ではない。それはトップレベルにおいてでさえ起こってはいけない理由はない。そこで全ての抽象的な機械のコピーを捨てて初期化するための明確な手続きが必要である。これをinit-amb として次のように定義する。

```
(macro init-amb (lambda (l)
  `(call-with-current-continuation
     (lambda (-c-)
       (%amb-catchers-stack% 'initialize)
       ((%amb-catchers-stack% 'push)
        (cons -c- #f))
       #t))))
```

さらに、amb によってコピーされた複数の機械が、元の機械も含めてどれも解に至ることができずにfail に至ってしまった場合に、例外的な処理を行うことができると便利である。手続き的にみれば、これはプログラムの実行経過のある地点を越えて後戻りを行おうとした場合に例外処理を起動するということになる。このような機構を Common Lisp のunwind-protect を真似て次のように実現する。

```
(macro amb-unwind-protect (lambda (l)
  (let ((exp1 (cadr l))(exp2 (caddr l)))
    `(call-with-current-continuation
       (lambda (-c-)
         ((%amb-catchers-stack% 'push)
          (cons -c- (delay ,exp2)))
         ,exp1)))))
```

実はamb とamb-unwind-protect は同じ定義である。

簡単な実行例を示そう。

```
(define (test-body)
    (let ((value (choice 10)))
        (write value)
        (newline)
        (fail)))

(define (test)
    (amb-unwind-protect
        (begin
            (display "protected-form")
            (newline)
            (test-body))
        (begin
            (display "cleanup-form")
            (newline))))
```

(if (init-amb) (test) 'end) を実行すると、まずprotected-form が表示され、続いて 10,9,...,1 が表示される。最後にcleanup-form が表示され終了する。

ここでトップレベルから(fail) を評価しようとするならば、その評価をせずに済むように(init-amb) が#f であったように振る舞う。すなわちend が返ってくるであろう。さらにもう一度(fail) を評価しようとするならば、その評価を避ける方法は存在しないので、評価が起こる。これはerror となりfail を評価してしまったというメッセージが出力されることになる。

A.4.3 全解収集

先に述べた 8-QUEEN の例で、(queen 8) を実行すると、92 個ある解の中の一つが得られる。それではその他の解を求めるにはどうしたらよいであろうか。コピーした機械はまだ存在しているから、トップレベルから強制的に(fail) を評価しようとすれば、その評価をしないで済むような別の機械 (系列が) 起動されることになる。これはちょうど Prolog のトップレベルにおいて';' を入力し、強制的にバックトラックさせることに似ている。このようにして次々に別の解を得ることができる。つまり全解探索の機能があるということになる。しかし、このままではその解の収集は評価結果を監視している人間が

行っているのであって、新たな解を求めるために先の解を求めた機械は捨てられてしまっている。また、あらかじめいくつの解が存在するのかを知ることができなければ、(fail) を強制的に何度行ってよいのかがわからないという問題もある。そこで、汎用の全ての解の集合 (リスト) を返すようなsetof を次のように定義することにする。

```
(macro setof (lambda (l)
  (let ((proc (cadr l)))
    `(let ((stack (make-stack)))
       (amb-unwind-protect
          (let ((ans ,proc))
            ((stack 'push) ans)
            (fail))
          (reverse
            (stack 'stack->list)))))))
```

ここで(setof (queen 8)) を評価すれば、全ての解のリストが得られる。

一般に一つの解を見つけるプログラムを定義するよりも、全ての解を得るプログラムを定義することの方がはるかに困難であることが多い。例えば Prolog には全解探索機能があるが、強制的にバックトラックを行う方法を用いると、Prolog の変数は代入ではなくユニフィケーションであるから、一度得られた情報は失われてしまい全解収集を行うことができない。しかしながら Prolog でもこの 2 階の述語であるsetof を実現することは可能である。それは副作用のあるassert 述語を用いて、大域変数への代入と同様なことが行えるからである。上記したsetof の定義も副作用を用いて実現されている。

ところで、これまで述べてきたスケジューリングによる問題はあるものの、その制約の中ではamb は並列プロセスへのfork であり、setof は全てのプロセスの終了を待ってその結果を受け取り、一つのプロセスへ渡すjoin であるとみなすことができる。

B マクロ機能を持たない Scheme のために

もしかしたら、あなたが利用できる Scheme の処理系にはマクロが備わっていないかもしれない。しかしその場合にもamb と同様のことが簡単に行える。amb をマクロとして定義しなければならない理由は、(fail) に出会ったためにやむを得ない場合を除いて、一方の引き数を評価して欲しくないからであった。ところで、特殊形式のif はこの条件を満たしている。つまり次のように定義しておけば、(amb exp1 exp2) の代わりに(if (flip) exp1 exp2) として同様のことが行える。

```
(define %amb-catchers-stack%
    (make-stack))

(define (flip)
  (call-with-current-continuation
   (lambda (-c-)
     ((%amb-catchers-stack% 'push)
      (delay (-c- #f)))
     #t)))

(define (fail)
  (if (%amb-catchers-stack% 'empty?)
      (error "FAIL evaluated.")
      (force
        (%amb-catchers-stack% 'pop))))
```

ただし、だからといって次のようにamb を定義することは誤りである。

```
(define (amb exp1 exp2)
  (if (flip) exp1 exp2))
```

これではamb を呼びだしたときに、exp1 とexp2 の両方の引き数が評価されてしまうからである。

C ISWIM から Idealized Scheme へ

Landin による関数型言語 ISWIM は、以下に示す文法の λ 算法の式で作られた値呼びの言語であった。

$$N ::= x \mid NN \mid \lambda x.N$$

ここで x は変数である。

ISWIM の操作的意味論は、Landin によって SECD マシンの項で定義された。Plotkin は、この定義が（部分的に）関数 $eval_v$ に等しいことを示した。（定数とその評価は無視することにする。）

1. $eval_v(V) = V$
2. $eval_v(MN) = eval_v(Q[V/x])$
   ただし、$eval_v(M) = \lambda x.Q$ かつ $eval_v(N) = V$ である。また、各々の V は値を表している。ここで値は変数であるか λ 抽象であると定義されている。

以下においては、メタ変数 $V, V_1, V_2, \cdots$ は変数を表すものとする。表記法 $M[N/x]$ は、M における x の全ての自由変数としての出現を N に置き換えることを示す。$(\lambda x.M)V$ の形をした式は、β_v-redex と呼ばれる。関数 $eval_v$ は、各々のステップにおいてλ抽象のスコープの内側ではなく、最左最外の β_v-redex を簡約するものとする。

Felleisen は、評価コンテクスト[4] の項において、この評価順序を形式化していた。ISWIM が評価するコンテクスト E は、次のように帰納的に定義される。

$$E ::= [\,] \mid EN \mid VE$$

ここで $[\,]$ は"hole"を表す。

もし E が評価コンテクストであるならば、$E[M]$ は E の hole において M を置いて得られる結果の項を示す。任意の閉じた項[5] M が値であるか $M = E[R]$ という形で表すことができることを容易に示すことができる。ここで R は β_v-redex である。また R はλ抽象の内側ではなく、M の最左最外 β_v-redex である。表記 $M \propto E[R]$ は $E[R]$ が M のこのユニーク表現であることを意味している。例えば、もしも $E_0 = (\lambda x.M)[\,]$ で、$E_1 = [\,]$ であるならば、以下のようになる。

$$(\lambda x.M)V = E_0[V] \propto E_1[(\lambda x.M)V]$$

ある評価コンテクストと β_v-redex の項における任意の値ではないもののユニーク表現は、コンテクスト書き換え規則を生む。

$$E[(\lambda x.M)V] \to_{\beta v} E[M[V/x]] \qquad (\to_{\beta v})$$

この反射推移閉包 $\to^*_{\beta v}$ は $eval_v$ に等しい。

定理 C.1

$$eval_v(M) = V \textbf{ iff } M \to^*_{\beta v} V$$

言い替えれば、$\to_{\beta v}$ は ISWIM の抽象化した操作的意味論を与えるということである。ISWIM の項 M は $M \to^*_{\beta v} V$ のとき、かつその時に限って V であると評価される。評価コンテクストの表記法は、項の表記中にその部分項が評価されるときの作法を明解に描き出す[6] 。

[4] コンテクストとは、ある一つの出現位置が空になっている項である。コンテクスト一般を $E[\,]$ で表す。$E[\,]$ は項のメタ表記であり、項そのものではないことに注意。

[5] 閉じた項とは、その項に含まれる自由変数の集合が空である項のことである。

[6] $\to^k_{\beta v}$ という表記は $\to_{\beta v}$ 規則の k 回の適用を示す。

定理 C.2

1. $E[M] \to^k_{\beta v} E[N]$ ならば $M \to^k_{\beta v} N$
2. $E[M] \to^*_{\beta v} V$ ならば $E[M] \to^* E[V_0] \to^*_{\beta v} V$ であるような V_0 が存在する。

次の評価のシーケンスにおける任意の i 番目について、

$$M_0 \to_{\beta v} M_1 \to_{\beta v} \cdots \to_{\beta v} M_i \to_{\beta v} \cdots$$

もし、$M_i = E[N]$ で、かつ N が値ではないとき、その部分項に影響する計算についての評価シーケンスを続ける前に、E はこの N を値に評価するのを"wait"しなければならない。それは E が評価された後になされる計算の残りの部分を表しているということである。コンテクスト E は評価シーケンスにおけるこのポイントにおいての N の継続（あるいは制御コンテクスト）と呼ばれる。評価コンテクストのこの表記は、以下でみるように、継続を扱うオペレータの操作的意味論の簡潔な仕様を与える。

Felleisen は ISWIM へコンテクストを操作する二つの構成子を組み込んだ Idealized Scheme あるいは IS と呼ばれる式を与えた[7] 。IS の式は ISWIM の文法を拡張することによって以下のように定義される。

$$N ::= \cdots \mathcal{A}(N) \mid \mathcal{C}(N)$$

オペレータ $\mathcal{A}$ と $\mathcal{C}$ は、それぞれ abort と control と呼ばれる。IS においては、任意の閉じた項 M は変数であるか $M = E[R]$ といったユニークに書き表されるものである。ここで R は β_v-redex か $R = \mathcal{A}(N)$ であるか $R = \mathcal{C}(N)$ であるかの何れかである。形式的ではなく述べると、$\mathcal{A}(M)$ の評価は M の評価において現在の評価コンテクストと継続を捨てる。このことは、コンテクスト書き換え規則として表現される。ここでコンテクストの評価の定義は、次のように明白な方法で IS の式が拡張されたものとなっている。

$$E[\mathcal{A}(M)] \to_{\mathcal{A}} M \qquad (\to_{\mathcal{A}})$$

$\mathcal{C}(M)$ の操作的意味論は、形式的ではないが、以下のように詳述できる。$\mathcal{A}$ と同じ様に、$E[\mathcal{C}(M)]$ の評価は制御コンテクスト E を捨てる。そして項 M は捨てられた制御コンテクストの手続き的な抽象に適用される。もしこ

[7] ここでは、Idealized Scheme の assignment-free な sub-language のみを取り扱う。

の手続きが任意のコンテクスト E_1 において値 V をもって呼び出されると、E_1 は捨てられ、$E[V]$ として評価を再開する。これは次の規則として表現される。

$$E[\mathcal{C}(M)] \rightarrow_{\mathcal{C}} M\lambda z.\mathcal{A}(E[z]) \qquad (\rightarrow_{\mathcal{C}})$$

オペレータ $\mathcal{A}$ は次のように $\mathcal{C}$ の項において定義され得る。

$$\mathcal{A}(M) \stackrel{def}{=} \mathcal{C}(\lambda d.M)$$

ここで d は M において自由ではないダミー変数である。

$$\begin{aligned} E[\mathcal{A}(M)] &= E[\mathcal{C}(\lambda d.M)] \\ &\rightarrow_{\mathcal{C}} (\lambda d.M)\lambda z.\mathcal{A}(E[z]) \\ &\rightarrow_{\beta v} M \end{aligned}$$

それゆえに、$\mathcal{A}(M)$ は定義された構成子として論じられ、規則 $\rightarrow_{\beta v}$ と $\rightarrow_{\mathcal{C}}$ は IS の操作的意味論の定義として論じられる。表記 $\rightarrow_u$ は二つの評価規則の和を示す。

$\mathcal{C}$ の操作的意味論は、`call/cc` のそれとは、$\mathcal{C}$ がその使用する位置へ戻ることを必要としないという点で異なっている。IS に`call/cc` を加えた変形を $\mathcal{K}$ と呼び、それは次の評価規則を持つことになる。

$$E[\mathcal{K}(M)] \rightarrow_{\mathcal{K}} E[M\lambda z.\mathcal{A}(E[z])] \qquad (\rightarrow_{\mathcal{K}})$$

しかし、この追加は、計算上 $\mathcal{K}$ に等しいオペレータが次のように定義できるので、必要ではない。

$$\mathcal{K}_d(M) \stackrel{def}{=} \mathcal{C}(\lambda k.k(Mk)) \qquad (\mathcal{K}_d)$$

$\mathcal{K}_d$ の一つの使用例として、Common Lisp の特殊形式 `catch/throw` [8] と同様のメカニズムをインプリメントすることがある。

[8] 訳者注: ただし、ここでいう`catch/throw` は Common Lisp のそれとは次のように若干文法が異なる。

Common Lisp: `(catch 'tag ... (throw 'tag val) ...)`
Scheme: `(catch tag ... (tag val) ...)`

余談ではあるが、SECDR-scheme では以下のように定義すればよいものである。

```
(macro catch
   (lambda (l) (let ((tag (cadr l)) (body (cddr l)))
      `(call/cc (lambda (,tag) ,@body)))))
```

この`catch` は、セクション A で示した Common Lisp と同様のものとは違って、`catch` 式の外側からでも起動することができる。それゆえ、`(catch x x)` は`(current-continuation)` に等しい。また、逆に`call-with-current-continuation` は、この`catch` を用いて以下のように定義できる。

```
(define (call/cc proc) (catch z (proc z)))
```

$E_0[\mathcal{K}_d(\lambda j.M)]$ の評価を現在のコンテクストに名前 j を用いてラベル付けする"catch"と考える。M の評価中に j が全く呼び出されないか"throw" されるならば、この式は"normally"を返す。一方、M の評価中に $E_1[jV]$ のように j の適用に出会うならば、値 V は j によってラベル付けされた位置へ"throw back" される。それは、コンテクスト E_1 が捨てられ $E_0[V]$ の残りが評価されることである。以下で Idealized Scheme の評価規則として、いかにこれが完結するかを示す。

もし $Q = \lambda z.\mathcal{A}(E_0[z])$ ならば

$$\begin{aligned} E_0[\mathcal{K}_d(\lambda j.M)] &\rightarrow_{\mathcal{C}} (\lambda k.k((\lambda j.M)k))Q \\ &\rightarrow_{\beta v} Q((\lambda j.M)Q) \\ &\rightarrow_{\beta v} Q(M[Q/j]) \end{aligned}$$

もしも $M[Q/j] \rightarrow^*_{\beta v} V$ ならば評価は以下について"normally"を返す。

$$\begin{aligned} &\rightarrow^*_{\beta v} QV \\ &\rightarrow_{\beta} \mathcal{A}(E_0[V]) \\ &\rightarrow_{\mathcal{A}} E_0[V] \\ &\cdots \quad \cdots \end{aligned}$$

一方、もし値が結局 throw されるならば、

$$\begin{aligned} Q(M[Q/j]) &\rightarrow^*_{\beta v} E_1[QV] \\ &\rightarrow_{\beta v} E_1[\mathcal{A}(E_0[V])] \\ &\rightarrow_{\mathcal{A}} E_0[V] \\ &\cdots \quad \cdots \end{aligned}$$

これはコンテクスト E_1 が捨てられて、復元されたコンテクスト E_0 において V の評価が続くことを示している。

参考文献

[1] Harold Abelson, Gerald Jay Sussman: Structure and Interpretation of Computer Programs, McGrawHill, 1985.

[2] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, Bruce F.Duba: Abstract Continuations: A Mathematical Semantics for Handling Full Functional Jumps, In Proc. 1988 Conference on Lisp and Functional Programming, pages 52-62, 1988.

[3] Timothy G.Griffin: A Formulae-as-Types Notion of Control, In Proc. 17th ACM Symposium on Principles of Programming Languages, pages 47-58, 1990.

[4] William Clinger, Jonathan Rees(Editors): Revised^4 Report on the Algorithmic Language Scheme, 2 November, 1991.

[5] Peyton Jones: The Implementation of Functional Programming Languages, Prentice Hall, 1987.

[6] Peter Henderson 著, 杉藤芳雄・二木厚吉 訳: 関数型プログラミング, 日本コンピュータ協会

[7] R. バード・P. ワドラー 共著, 武市正人 訳: 関数プログラミング, 近代科学社

[8] 井田哲雄: 計算モデルの基礎理論, 岩波書店

[9] 木村泉・米澤明憲: 算法表現論, 岩波書店

[10] 高橋正子: 計算論 ——計算可能性とラムダ計算, 近代科学社

[11] 淵一博・黒川利明 編著: 新世代プログラミング, 共立出版