# アクションセマンティクスによる Scheme の意味記述*

1G－9

小村 昌弘　伊藤 貴康†
東北大学 大学院 情報科学研究科‡

## 1 はじめに

公理的意味論や表示的意味論が様々な言語に対して与えられているが、言語処理系の作成者にとって分り易く実用的な形式的意味論が求められている．そこで言語の操作的概念を直接表現し、英語で表記することで分り易さを実現したアクションセマンティクス [1] が P.Mosses によって提案されている．

本稿ではアクションセマンティクスを用いて Scheme[2] の形式的意味を与え、伝統的な表示的意味論と比較する．また、アクションセマンティクスによる記述が正しく動作することを確認するために ASI(Action Semantics Interpreter) を試作した．

## 2 アクションセマンティクス

アクションセマンティクスは言語の意味を操作的な意味を持つアクションによって表示する．

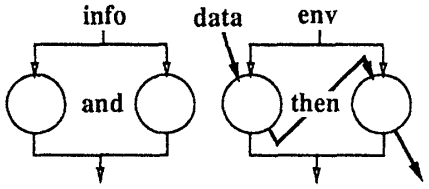
アクションはプリミティブアクションとアクション結合子から構成される．プリミティブは情報(データ・環境・記憶) を操作する．結合子はアクションを結合するとともに、アクション間の情報の流れを表現する．

### 2.1 プリミティブアクション

| | |
|---|---|
| give $d$ | データ $d$ を与える． |
| bind $t$ to $d$ | トークン $t$ をデータ $d$ に束縛する． |
| store $t$ in $c$ | データ $d$ をセル $c$ に格納する． |
| enact $d$ | 抽象 $d$ を実行する． |

### 2.2 アクション結合子

| | |
|---|---|
| A or B | A・B のどちらかを実行する． |
| A and B | A・B を実行し、結果をまとめる． |
| A then B | A・B を逐次的に実行する． |




## 3 Scheme の意味記述

まず、Scheme の最も特徴的な機能であるコンティニュエーションをアクションセマンティクスで操作する方法を示す．

### 3.1 コンティニュエーション

コンティニュエーションは計算のある時点での残りの計算を表し、非局所的脱出、コルーチンなどの強力な制御機能を実現する．

アクションセマンティクスではこのコンティニュエーションをアクションのコンテキストとして取り出すことができる．例えば、次のアクションの A におけるコンテキストは、A を A の結果を受け取るアクションに置き換えることで得られる．

A then B → give the given value then B

表示的意味論で A then B は次のようになる．ここで $\rho$ は環境、$\kappa$ はコンティニュエーションである．

$\lambda\rho\kappa.\mathrm{A}\rho(\lambda\epsilon^*.\mathrm{B}\rho\kappa)$

### 3.2 Scheme の構文の意味

次の Scheme の基本構文を考える．

```
Exp  →  K | I | (E0 E*)
     |  (lambda (I*) E*)
     |  (if E0 E1 E2)
     |  (set! I E)
```

式 E のアクションセマンティクス evaluate$[\![E]\!]$ は次のように与えられる．

```
evaluate [[K]] = give value [[K]]
evaluate [[I]] =
  give value stored in cell bound to I
evaluate [[(lambda (I*) E*)]] =
  give closure of abstraction of
      | | rebind
      | | moreover bind-parameter* I*
      | hence
      | | evaluate [[E0]] then ···
      | |   then evaluate [[Em]]
  (where E* = E0 ··· Em)
```

定数・識別子・ラムダ式はコンティニュエーションに値を渡す．これは give アクションで表現される．ラムダ式の本体は式の並びを then で結合する．

```
evaluate [[(E0 E*)]] =
  | evaluate* [[E0E*]]
  then enact application of given function#1
                          to given value/1
evaluate* [[E0E*]] =
  evaluate [[E0]] and ··· and evaluate [[En]]
  (where E* = E1 ··· En)
```

---

*Action Semantics of a subset of Scheme

†Masahiro KOMURA, Takayasu ITO
‡Department of Computer and Mathematical Sciences, Graduate School of Information Sciences, Tohoku University

関数適用は複合式を評価し、関数を引数に適用する. 複合式はそれぞれの評価を and で結合する.

```
evaluate ⟦(if E0 E1 E2)⟧ =
  | evaluate ⟦E0⟧
  then | | check (not given value)
       | and then evaluate ⟦E2⟧
       or | check (given value)
            and then evaluate ⟦E1⟧
```

if 文の選択文は or で結合され、check アクションによりどちらかが実行される.

```
evaluate (set! I E) =
  | evaluate ⟦E⟧
  then | store given value in cell bound to I
       then give unspecified
```

上記の Scheme の基本構文に対する表示的意味を付録に与えたが、それと比較した時にアクションセマンティクスはどうような操作を行なうのかを明示的に表現しているので、これからコンパイルコード生成が容易に行なえる.

### 3.3 call/cc と call/ep の意味

call/cc は一引数の関数を引数にとり、適用時の残りの計算であるコンティニュエーションを関数として取り出し、それを引数に渡す関数である.

```
call/cc =
  abstraction of
  | enact application of given function
  |                    to current continuation
current continuation =
  abstraction of | ⟨context⟩
                 then escape
```

call/cc のコンティニュエーションはコンテキストを実行し脱出する抽象になる.

処理系での実行効率の向上を目的に、call/cc のコンティニュエーションの働きを非局所的脱出に制限したのが、call/ep[3] である.

call/ep のコンティニュエーションは適用されるとタグが示す制御ポイントまで戻る. 表示的意味論ではこれを簡単に表現できないが、アクションセマンティクスでは次のように表現できる.

```
call/ep =
  abstraction of
  | | enact application of given function
  | |                  to abstraction of
  | |                     | | give tag
  | |                     | and give given value
  | |                     | then escape
  | trap | | check (given value#1 is tag)
  |      | and then give given value#2
  |      or | check (not (given value#1 is tag))
  |         and then escape
```

call/ep のコンティニュエーションはタグを付けて脱出する抽象になる. この脱出は同じタグを持つトラップアクションで捉えられる.

## 4 ASI(Action Semantics Interpreter) の試作

アクションセマンティクスは分り易さを求めたが、その記述は冗長になり、記述の正しさの確認も必ずしも容易ではない. そこで、カーネルアクションを入力に取りそれを実行して結果を導く ASI を試作した.

ASI ではデータを組で表わし、環境をトークンとセルの組の集合で表わし、記憶をセルと値の組の集合で表わす. アクションは直接これらを操作する.

ASI への入力の負荷を軽減するため、Scheme の基本構文をアクション系列に変換するコンバータを用意した.

### 4.1 実行例

Scheme の組込関数を含む初期環境をアクションの環境として用意してから実行する.

```
> (built-in)                    .. 初期環境の用意
> (convert '(not #t))           ..Scheme 式の変換
(then (and (give (at sto (at env not)))
           (give true))
      (enact (then (provision (rest them))
                   (component 1 them))))
> (perform)                     .. アクションの実行
(completed false ())
```

この他にもいくつかの例で、記述が意図した動作をすることを確認した.

### 付録

$\mathcal{E}⟦\mathrm{K}⟧ = \lambda\rho\kappa.\kappa\langle\mathcal{K}⟦\mathrm{K}⟧\rangle$

$\mathcal{E}⟦\mathrm{I}⟧ = \lambda\rho\kappa.\lambda\sigma.\kappa\langle\sigma\rho\mathrm{I}\rangle\sigma$

$\mathcal{E}⟦(\texttt{lambda}\,(\mathrm{I}^*)\,\mathrm{E}^*)⟧ =$
$\lambda\rho\kappa.\kappa\langle\lambda\epsilon^*\kappa'.\lambda\sigma.\mathcal{E}⟦\mathrm{E}_0⟧\rho'(\lambda\epsilon^*.\cdots\mathcal{E}⟦\mathrm{E}_m⟧\rho'\kappa'))\sigma'\rangle$
$(where \rho' = \rho[\alpha/\mathrm{I}]\cdots, \sigma' = \sigma[\epsilon/\alpha]\cdots)$

$\mathcal{E}⟦(\mathrm{E}_0\,\mathrm{E}^*)⟧ =$
$\lambda\rho\kappa.\mathcal{E}(\langle\mathrm{E}_0\rangle\S\mathrm{E}^*)\rho(\lambda\epsilon^*.(\epsilon^* \downarrow 1)(\epsilon^* \dagger 1)\kappa)$

$\mathcal{E}^*⟦\mathrm{E}_0\mathrm{E}^*⟧ =$
$\lambda\rho\kappa.\mathcal{E}⟦\mathrm{E}_0⟧\rho(\lambda\epsilon_0.\cdots\mathcal{E}⟦\mathrm{E}_n⟧\rho(\lambda\epsilon_n.\kappa\langle\epsilon_0\cdots\epsilon_n\rangle))$

$\mathcal{E}⟦(\texttt{if}\,\mathrm{E}_0\,\mathrm{E}_1\,\mathrm{E}_2)⟧ =$
$\lambda\rho\kappa.\mathcal{E}⟦\mathrm{E}_0⟧\,\rho\,(\lambda\epsilon.\epsilon = \mathit{false} \rightarrow \mathcal{E}⟦\mathrm{E}_2⟧\rho\kappa, \mathcal{E}⟦\mathrm{E}_1⟧\rho\kappa)$

$\mathcal{E}⟦(\texttt{set!}\,\mathrm{I}\,\mathrm{E})⟧ =$
$\lambda\rho\kappa.\mathcal{E}⟦\mathrm{E}⟧\rho(\lambda\epsilon.\lambda\sigma.\kappa\langle\mathit{unspecified}\rangle\sigma[\epsilon/\rho\mathrm{I}])$

$\mathit{call/cc} = \lambda\epsilon\kappa.\epsilon\langle\epsilon^*\kappa'.\kappa\epsilon^*\rangle\kappa$

### 参考文献


[1] Peter D. Mosses. Action Semantics. Cambridge University Press, 1992.

[2] William Clinger and Jonathan Rees. Revised$^4$ Report on the Algorithmic Language Scheme. 1991.

[3] 清野 智弘. 伊藤 貴康. PaiLisp の並列構文の実現法と評価. 情報処理学会論文誌, 第 34 巻, 第 12 号. 1993.