数理解析研究所講究録
第458巻 1982年 256-265

denotational semantics を使った万能interpreter

東工大　情報科学　江崎武史

1. はじめに

従来、プログラム言語を記述する方法として、構文をバッカス記法で示し、その意味を自然言語で説明する方法がとられてきた。構文をバッカス記法で形式的に定義することにより、記述の正確さ・簡潔さが得られ、それに対する種々の構文解析の技法も蓄積されている[1]。もし意味も含めてプログラム言語全体を形式的に定義できれば、ソフトウェア生産性の向上に役立つことが期待され、更に、コンパイラの自動生成やプログラムの正当性の証明の可能性が考えられる。

プログラム言語の意味を形式的に定義する方法は、これまで種々考えられており、たとえば次のものがある[2]。

- W-grammar
- axiomatic semantics
- VDL (operational semantics)

・attribute grammar

・denotational semantics

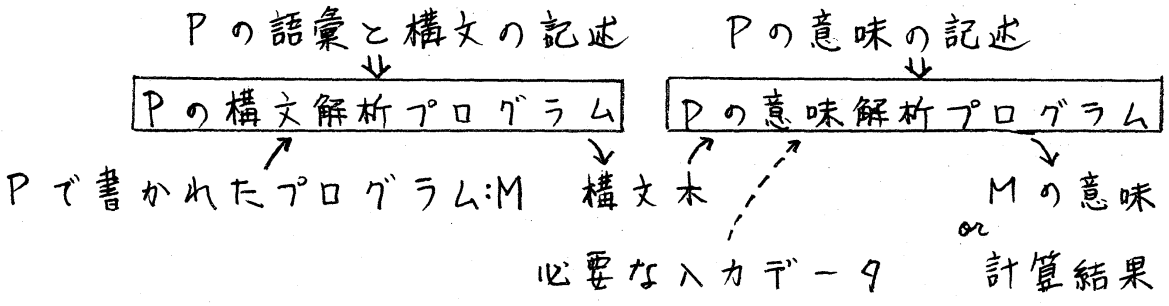
これらの中で、denotational semantics は正しい構文構造をもつプログラムの「意味空間」をあらわすdomain を定義し、プログラムと意味との対応を関数形式であらわす。この方法は、次のような点ですぐれていると思われる[3]～[6]。

(i) 汎用性。理論的には、$\mathbb{N}$ 上の任意の computable function が記述できる。また operational semantics や attribute grammar などによる記述も denotational semantics で書ける。

(ii) スコット理論により数学的基礎が確か。

(iii) プログラムと意味との対応をあらわす関数はLISP で自然な形で書け、与えられたデータに対してこれを実行することにより interpret できる。

このようにして、任意のプログラム言語（たとえばP）のdenotational semantics を使った形式的定義を与えると、Pで書かれたプログラムを interpret することができる。これが万能interpreterの原理である。



（必要な入力データを入れなければMの意味がLISPの関数として出力される。）

2、チューリング機械をシミュレートする
プログラム言語TURING

例として、チューリング機械をプログラム言語とみなし、その形式的定義とプログラムを与えて意味（＝実行結果）を出力することを考えよう。対象とするチューリング機械は、右側のみ無限のテープを1本持つ決定性の機械である。

(2.1) TURINGの語彙と構文の記述

(i)語彙の記述

〈テープ用文字〉 ::= 〈英字〉 | b　　（bは空白を示す）

〈状態〉 ::= 〈数字〉(〈数字〉)*

他に特殊記号として { } , [ ] ← →

(ii)構文の記述

〈チューリング機械〉 ::= 〈開始状態〉,〈終了状態の集合〉,〈遷移関数〉

〈開始状態〉 ::= 〈状態〉

〈終了状態の集合〉 ::= {〈状態〉(,〈状態〉)*}

〈遷移関数〉 ::= {〈四ツ組〉(,〈四ツ組〉)*}

〈四ツ組〉 ::= [〈状態〉,〈テープ用文字〉,〈状態〉,〈テープ用文字〉]

| [〈状態〉,〈テープ用文字〉,〈状態〉,←]

| [<状態>, <テープ用文字>, <状態>, →]

たとえば、(i), (ii)から次のプログラムが書ける。

1, {2}, {[1, a, 1, →], [1, b, 2, a]}

このプログラムがあらわすチューリング機械は、状態1で実行を開始し、aを読み飛して初めて出会った空白をaに書き換える。

(2.2) TURING の意味の記述

右側のみ無限のテープは、左端の欄から順に 0, 1, … と非負整数を対応させれば、非負整数からテープ用文字への関数とみなせる。即ち、

$Num = \{0, 1, \cdots\}$, $Alpha = \{a, b, \cdots, z, b\}$

$Tape = Num \to Alpha$ $(= \{f : Num \to Alpha\})$

とすれば、テープは $Tape$ の元である。

我々は無限の長さのテープを実際の文字列として取扱うことができない。そこで、ここではチューリング機械の入力テープを指定するのに、入力データとして左端からの有限部分を文字列で与え、その右側には空白がつまっているものとする。また、出力テープの左端から右へ最初の空白に出会うまでの有限部分を、出力データとして知らせてくれるものとする。また、ヘッドが左端からさらに左へ動こうとした時や遷移が未定義の時は、エラーを知らせてくれるものとする。従

って、入力データの domain *Input* と出力データの domain *Output* は、次のようになる。

Input = *Alpha**, *Output* = (*Alpha* − {b})* + {error}

TURINGプログラムの syntactic domain を、*Turing* とする。

Turing = {w | <チューリング機械> ⇒ w}

プログラムの意味をコンパイラ的に考えると、そのプログラムをコンパイルした結果のオブジェクト・プログラムが、その意味と考えられる。オブジェクト・プログラムは、入力データが与えられると実行を開始して、結果として出力データを返す関数（即ち *Input* → *Output* の元）と考えられる。従って、プログラムの意味を与える関数 Turing は *Turing* から *Input* → *Output* への関数となる。一方、インタプリタ的に考えると、プログラムは入力データとのペアで意味が考えられ、その意味はプログラムを入力データで実行した結果の出力データと考えられる。従って、プログラムの意味を与えるもう1つの関数 Turing′ は *Turing* × *Input* から *Output* への関数となる。

関数 Turing は、たとえば次のように定義できる。

Turing [M] : *Input* → *Output*

λinput. Output [Move [Mの開始状態, Input [input], 0]]

ただし

Move [s, t, n] = t'

(s, t, n) $\xrightarrow[M]{*}$ (s', t', n')　　(s' は終了状態)

Input : λカデータをテープに変換する

Output : テープを出力データに変換する

3. denotational semantics 記述用メタ言語 LAMBDA

このシステムでは denotational semantics を記述するためのメタ言語として、次のものを用意してある。

(1) domains

(1.1) syntactic domains

語彙や構文の非終端記号に対応した，構文木の集合

(1.2) primitive semantic domains

(i) $\mathcal{N}um$ = {整数}

(ii) $\mathcal{B}ool$ = {true, ~~false~~}

(iii) $\mathcal{S}trcon$ = {文字列}

(iv) $\mathcal{E}rror$ = {エラーメッセージ}

(1.3) semantic domain construction

(i) $\mathcal{D} = \mathcal{D}_1 \to \mathcal{D}_2$

(ii) $\mathcal{D} = \mathcal{D}ummy \to \mathcal{D}_1$

(iii) $\mathcal{D} = \mathcal{D}_1 + \cdots + \mathcal{D}_n$

(iv) $\mathcal{D} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_n$

(v) $\mathcal{D} = \mathcal{D}_1^*$

あるいは、これらで構成される方程式の最小解

(2) functions

(2.1) primitive functions

(i) doman 用基本関数

(ii) Equal : $\mathcal{D}_1 \times \mathcal{D}_2 \to Bool$

(2.2) constant

(i) 0, 1, ⋯ $\in Num$

(ii) true, false $\in Bool$

(iii) 文字列 $\in Strcon$

(iv) nil $\in \mathcal{D}^*$

(2.3) function construction

(i) $\sigma[\tau_1, \cdots, \tau_n]$

(ii) (イ) $\lambda d_1, \cdots, d_n . \sigma$

(ロ) $\lambda . \sigma$

(iii) if σ_1 then τ_1 elif σ_2 then τ_2 ⋯ else τ_n fi

(iv) (イ) let $id_n = \sigma$ in τ

(ロ) letrec $id_n = \sigma$ in τ

(v) case σ of when $\sigma_{i_1}, \cdots, \sigma_{in_1} : \tau_1$ ⋯ when others : τ_k

4. おわりに

(1) 構文解析の道具として、次のものを使った。

LEXGEN (scanner generator)[7]

YACCO (LALR(1) parsing table generator)[8]

(2) 付録に TURING プログラムの interpret の例を示す。

(3) データ構造として integer と Boolean を持ち、Algol 風の制御構造を持つ小さなプログラム言語 SMALL を考え、万能 interpreter で実行させた。この言語の構文規則の数は 25, 意味関数の数は 43 になった。

(4) LISP での関数はすべてリスト構造という universal domain 上での関数になり、domain の情報は活用されず、type check がおこなわれない。この点での改良が今後望まれる。

謝辞

適切な助言・御指導をいただいた高橋正子助教授, 山崎秀記助手に深く感謝致します。

参考文献

[1] A.V. Aho & J.D. Ullman : The Theory of Parsing, Translation and Compiling, Prince-Hall Inc. (1972)

[2] M. Marcotty, H.F. Ledgard, & G.V. Bochmann : A Sampler of Formal Definitions, Computing Surveys, Vol. 8, No. 2, June (1976)

[3] J.E. Stoy : Denotational Semantics : The Scott-Strachey Approach to Programming Languages, The MIT Press (1977)

[4] 高橋正子：スコット理論，情報処理 Vol. 20, No. 11 (1977)

[5] R.D. Tennent : The Denotational Semantics of Programming Languages, CACM Vol. 19, No. 8 (1976)

[6] M.J.C. Gordon : The Denotational Description of Programming Languages, Springer-Verlag (1970)

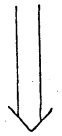
[7] 佐々，徳田，篠木，井上：Design and Implementation of a Multpass-Compiler Generator, Reserch Reports on Information Sciences No. C-24 (1979) 情報科学科，東工大

[8] 小野典彦：あいまいな文法を許す Parser Generator の作成，卒業論文 (1979) 情報科学科，東工大

TURING プログラムの interpret の例

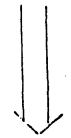
```
1,<2>,<<1,A,1,>>,<1,@,2,Z>>
```

--- プログラム



```
( 1 ( 2 1 ) , ( 3 < ( 5 2 ) > ) , ( 6 < ( 7 ( 8 ( 11
  < 1 , A , 1 , > > ) ) , ( 9 < 1 , @ , 2 ,
Z > ) ) > ) )
```

--- 構文木



```
((TURING_MACHINE_SEM (QUOTE
( 1 ( 2 1 ) , ( 3 < ( 5 2 ) > ) , ( 6 < ( 7 ( 8 ( 11
  < 1 , A , 1 , > > ) ) , ( 9 < 1 , @ , 2 ,
Z > ) ) > ) )
)) '(A A A))

==> (A A A Z)
```

--- 入力データ (A A A) を与えて interpret