第14卷 第3期 郑州工学院学报 Vol.14 No.3
1993年 9 月 Journal of Zhengzhou Institute of Technology Sep. 1993

从 DS 到 INTERPRETER 的机械转换*

——面向语义的软件自动生成

紫玉梅

(郑州工学院计算机与自动化系)

摘 要: 指称语义(*DS*)是描述程序设计语言语义的一种强有力的工具，抽象级别较高，难于在机器上实现，解决的办法就是将 *DS* 转换成易于在机器上实现的语言程序。本文给出了 *DS* 到解释程序（*INTERPRETER*）的机械转换方法，从而将语言的形式化定义与机器实现有机地结合起来，本文的工作是在面向语义的软件自动生成领域的进一步探索。

关键词: 指称语义，*Scott*—域，解释程序

中图分类号: TP31

1 DS 到 INTERPRETER 的转换算法

描述指称语义[1][2]需要一种形式化的语言，称为指称语义定义语言(DSDL)。本文的 DSDL 是以 Scott-Lambda 为基础 Scott 论域上的一种类型化的 Lambda 语言。DS 到 INTERPRETER 转换就是要将 DS 的各个部分转换成 INTERPRETER 的各个部分。

转换框图如图 1。

其中: Control Routine（控制程序）包括如下四部分:

- 接收程序。
- 调用 Syntactic Analysing Routines (语法分析程序）产生执行树。
- 调用语义函数作用于执行树。

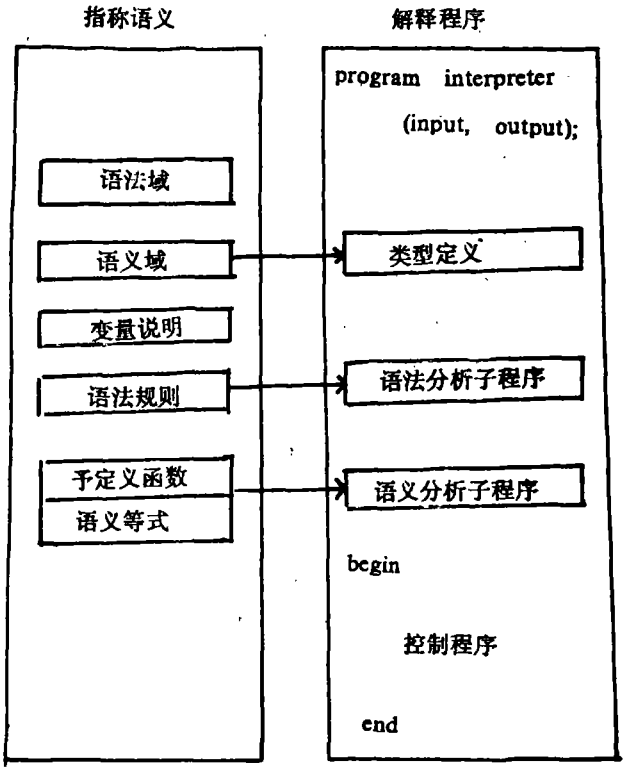


图 1

* 收稿日期: 1992-12-03

·输出作用结果.

INTERPRETER 是用 Pascal*语言来书写的. Pascal*可非形式化地描述为:

Pascal*是对 Pascal[3]稍加扩充得到的. 它允许表达式的值类型和函数返回值类型是多元组(记录类型数据), 允许类型说明在后, 使用在前, 并引进条件表达式. 除此以外, 其它与 Pascal[3]相同.

由转换框图可得 DS 到 INTERPRETER 的转换算法如下:

step1. Syntactic Domains (语法域) 不产生 Pascal 代码.

step2. Syntactic Rutes (语法规则) 转换成语法分析子程序.

step3. Semantic Domaims (语义域) 转换成 Pascal 的 TYPE 定义

step4. Variable Declarations (变量说明) 不产生 Pascal 代码, 但用于构造环境, 以便求语义表达式的类型.

step5. Predefining Functions (予定义函数) 转换成语义分析子程序.

step6. Semantic Equations (语义等式) 转换成语义分析子程序.

下面将给出上述算法中主要步骤的转换方法.

2 Syntactic Rules 到语法分析子程序的转换

指称语义到解释程序的转换需要把语法规则转换成语法分析子程序, 这一过程即自动生成语法分析程序的过程. 这方面的工作目前已比较完善, 如 YACC 等. 所以, 详细过程在此不予讨论.

为了整个过程的完整性, 这里介绍一下执行树.

语法树如同三元式、四元式一样, 是执行语法分析程序产生的中间代码, 树上的结点可以带有一定的语义执行时所需的信息. 因此, 把这样的语法树又称为执行树.

执行树的 Pascal 定义:

```
tree = ↑ node
node = RECORD
       case      t: nodetype oF
          t1:  (sid1S1   :D1);
          t2:  (sid2S2   :D2);
          ⋮
          tm:  (sidmSm   :Dm);
          tm+1: (sidm+1Sm+1·····sidm+1Sm+1Km+1: tree);
          ⋮
          tn:  (sidnSn·····sidnSnkn: tree);
          end
modetype = (t1, t2, ··· tn)
```

标志域 t_i ($1<i<m$)对应的节点为叶节点;

标志域 t_i ($m+1<i<n$)对应的节点为中间节点。

域名 Sid_iS_{ij} 就取为: 标志域为 t_i 时所定义的语法物的子语法物名。

3 语义域Domains到Pascal类型的转换

DS中的Domains定义与Pascal程序中的TYPE类型说明部分是一致的，因此，要实现DS到INTERPRETER的转换必须首先将DS中的Domains定义转换成Pascal的TYPE定义部分.

转换规则为:

```
domdef  d_i d_1 = Dom1                    Type  dtyp〖Dom1〗↓1;
        d_i d_2 = Dom2   ⇒                      d_i d1 = dtyp〖Dom1〗↓2;
          ⋮                                          dtyp〖Dom1〗↓3;
        d_i dk = Domk                             ⋮
domend                                          dtyp〖Domk〗↓1;
                                                didk = dtyp〖Domk〗↓2;
                                                     dtyp〖Domk〗↓3
```

其中 dtyp: Domain→TYPE× TYPE× TYPE

例:dtyp〖did〗=(—, did,—)

```
dtyp〖D1*〗=(dtyp〖D1〗↓1, ↑seqrec,
                seqrec = Record
                                        elm: dtyp〖D1〗↓2;
                                        next: ↑seqrec
                                        End: dtyp〖D1〗↓3)
```

转换例:

```
domdef IDE = char                  Typeide = char; val = integer;
       VAL = integer                        state = ↑ maprec;
       STATE = IDE  m  VAL ⇒                  maprec = Record
domend                                        dom: ide;
                                              rang: val
                                              next: ↑ maprec
                                              End;
```

4 予定义函数及语义等式到语义分析子程序的转换

予定义函数 $f(x_1, \cdots, x_k)=e$ 中的 e 的书写同语义等式的右端，所以这里只讨论语义等式到语义分析子程序的转换. 转换算法如下:

设有kk个语义函数.

```
For   i=1  To  kk  Do
Begin
```

sidi: M→D→…D→D

$sidi〖sidim_1〗=\lambda p_1 p_2 \cdots p_k \cdot e_1$

sidi〖sidim$_2$〗 $=\lambda p_1p_2\cdots p_k\cdot e_2$

⋮

sidi〖sidim$_n$〗 $=\lambda p_1p_2\cdots p_k\cdot e_n$

⇒

```
Func  sidi (sidist: tree, pL:T1,…pk: Tk):T;
                SPC1
                SPC2
                ⋮
                spcn
    Begin Case sidist↑ t of
            t1: sidi=FUN1;
            t2; sidi=FUN2;
               ⋮
             tn: sidi=FUNn
      End
  End
```

其中(SPCj, FUNj)=lpac(ej), (j=1,2,…,n)。

函数 lpac 由以下步骤完成:

lstep1. 构造 Lambda 树

lstep2. 实施 Lamdda 变换

lstep3. 生成 Pascal 代码

lstep4. 对 lstep3. 产生的$\begin{cases} CODE(a) \\ e \end{cases}$执行:

spcj←CODE(a); FUNj←e。

4.1 Lambda 树 tree

Lamdda 树是为方便实现 Lambda 表达式到 Pascal 程序的转换而引进的工具，是自上而下的无循环图，每一个 Lambda 表达式对应一个 Lambda 树。其定义如下:

tree: ExP→LTREE

LTREE=FLAG× LTREE× LTREE× LTREE

UVID U CONS U FID

例: tree〖λvid.E〗=(abs, vid, tree〖E〗,–)

tree〖let vid =E1 inE〗 = (let, vid,

tree〖E1〗,tree〖E〗)

4.2 Lambda 变换

Lambda 变换是为方便代码生成而对 Lambda 树实施的一系列等价变换，有 LET 变换、WHERE 变换、TUPLE 变换、APPLY 变换、ABSTRACT 变换、FULL 变换等。

如 LET 变换:

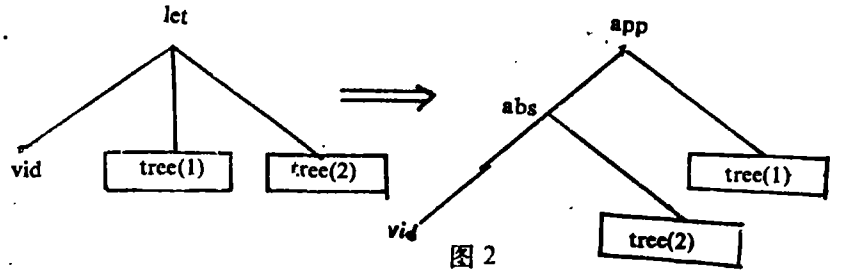


图 2

ABSTRACT 变换:

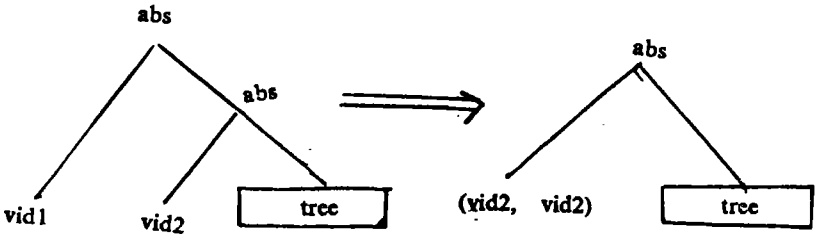


图 3

4.3 代码生成

代码生成是通过对经过变换的 Lambda 树自下而上进行变换完成的。每次按某一规则的左部转换 Lambda 树的最低层基本树型，产生相应的 Pascal 代码。直到 Lambda 树的标志为 null，代码生成过程结束。

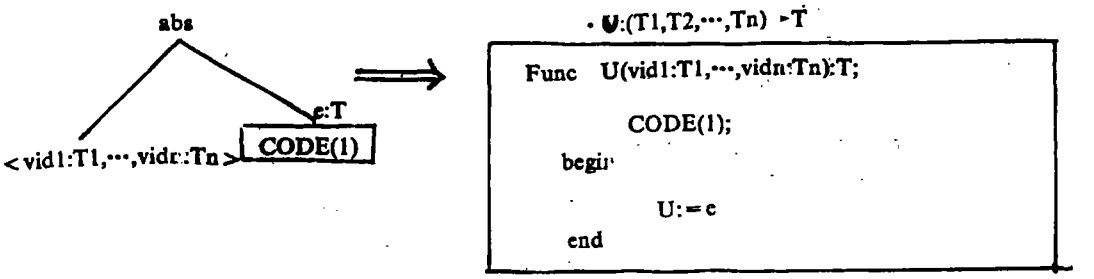
下面的规则表示，若当前 Lambda 树的最底层树型为⇒的左部，则将产生右部的代码，同时树型变为一个节点。Lambda 树的叶节点上的标的域名 T 是由 DS 中的 Variable Declarations 求得的。定义不同的函数时使用的名字 U 每次都对应一个新名字，因而不会出现同名问题。

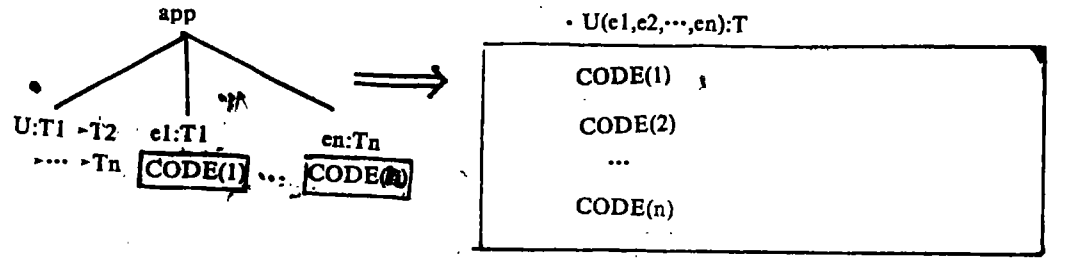
转换规则共二十多个，这里只给出几个作为示例:

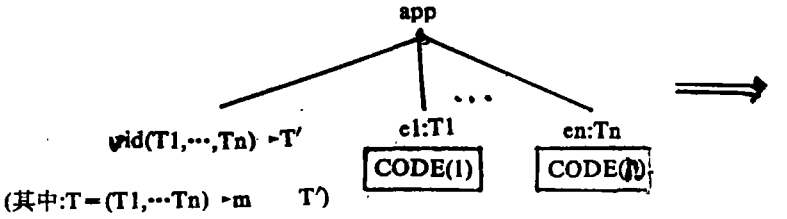
5 讨论

我们已使用一种非常普遍而应用广泛的语言 Pascal 使指称语义得以编码，文中给出的转换方法适用很大一类语言，同时产生的代码在结构上与人工开发的比较接近。从程序转换的角度，这里是将抽象的高阶函数定义的 Lambad 语言转换成可实现的过程式语言的转换提供一点借鉴，还有待于进一步研究，本文的工作也为面向语义的软件自动生成技术的研究提供了又一可行的途径。

本文要做的进一步工作是证明文中方法的正确性并在机器上实现。这一工作可采用的方法是将书写算法的元语言进一步形式化，然后通过它把源—指称语义（DS）与目标—解释程序（Pascal 程序）联系起来。这种方法之所以可取不仅是因为它可以以一般的或独立于语言的方式证明算法的正确性，而且由于有了形式化的转换有助于完成它的功能的自动实现。







· U(vid,e1,…,en):T′

```
Func   U(x:T, t1:T1,…,tn:Tn):T′;
    var   p:T;
    begin  p:=x;
           while   not ( (p ▸. dom1=e1)
                     and  (p ▸.dom2=e2)
                     …
                     and (p ▸.domn=en))
             do    p:=p ▸.next;
                   U:=p ▸.range
    end;
    CODE(1);        …      CODE(n)
```

图4

参 考 文 献

〔1〕金成植. 形式语义学和程序正确性, 吉林大学计算机系. 1986年2月

〔2〕周巢尘. 形式语义学引论. 计算机研究与发展. 1985年7月, 1985年8月

〔3〕王诚等. PASCAL程序设计及其应用. 清华大学出版社, 1983

〔4〕Michael J.G. Gordon, THE DENOTATIONAL DESCRAPTION OF PROGRAMING LANGUAGE, Spings-Verlag, 1979.

〔5〕Peter D. Massess, SIS-Scmantics Implementation System, Compiler Generation Using Donotational Semantics, lectur Noticesin Computer Science, Vol.45, P436-P441.

〔6〕Uwe Kastens, GAG: A Practical Compiler Generater, Lecture Notices in Computer Science, Spring-Verleg, 1982

〔7〕KARL-JOUKO RAIHA, Rerised Report on the Compiler Writing System HLP78, University of Holsinki, 1983.

〔8〕KAIHA, Attr: bute Grammar Design Using the Compiler Writing System HLP, Methods and Tools for Compiler Comstruction. Cambridge University Press, 1984, P183-P206.

The Inflexible Transformation from DS to Interpreter

Chai Yumei

(Zhengzhou Institute of Technology)

Abstract: Denotational semantics (DS) is a strong and high abstractive tool which represents programing language's semantics. It is difficult to implement. The solution to tramsform from DS into program which is realized easily. This paper represents a inflenible transformation from DS to Interpreter. It integrate formal definition of programing language and machine implementation closdy. The work in the paper is a deaper seek in field of sema-tics-direct automation producing softivacl.

Keywords: Denotational Semantics, Scotl-Domain, Interpreter