程序变换在程序语言中的一种表示
——兼论变换型语言

张乃孝

(北京大学计算机科学技术系,北京 100871)

A NOTATION OF PROGRAM TRANSFORMATION IN PROGRAMMING LANGUAGES— ON TRANSFORMATIONAL PROGRAMMING LANGUAGES

Zhang Naixiao

(*Department of Computer Science and Technology, Peking University, Beijing* 100871)

Abstract In this paper, the notion of transformational programming language is introduced. The new language constructs— "transformation module" and "transformation control directive" are defined. They play the key roles in the transformational programming languages. We explain that a transformation module can be used to describe a partial implementation of an abstract data type and a transformation control directive can be used to apply a transform defined by a transformation module to an abstract variable and the related operators in a program. Some of the important issues, such as abstraction, expressiveness and flexibility of representation in the transformational programming language and correctness of the transformational program are discussed in the later part of this paper.

摘要 本文首先引入了"变换型语言"的概念,给出了代表这种语言特征的机制:"变换模块"和"变换控制命令"的具体定义;举例说明了如何使用"变换模块"描述一个抽象数据类型的部分实现,并通过"变换控制命令"来完成程序中抽象变量及有关操作的变换过程;最后,讨论了变换型语言表示的抽象性,一般性和控制的灵活性,以及变换型程序的正确性等问题.

§1. 程序变换与变换型语言

程序变换的基本目标是实现一个程序到另一个程序的等价转换.[15]近十几年来,程序变换一直是计算机研究的重要方向.目前,这方面的研究已经将变换的思想扩展到从软件的

本文1991年3月14日收到,1991年6月19日定稿.本项研究得到了国家自然科学基金的资助.作者张乃孝,教授,主要研究领域为软件及软件理论.

规范说明到代码生成的各个软件研制阶段，考虑从非常高级的规范说明语言到可执行的过程语言描述之间一系列的转换问题. 我们可以用公式：Specification＋Transformation＝Software 来概括这项研究的主要目标.

目前这类通用且有效的软件系统仍处于研究中，分析其困难主要来自两方面：一是要定义一种通用的广谱语言，用以描述各种类型和抽象层次的软件；另外还要总结出所有相同和不同的层次上描述的程序之间转换的规则，刻划出这种语言的源程序之间的语义的等价关系. 现实的权宜之计是开发各种具体领域的程序变换系统，这样规范描述语言的选择和变换规则的提取就相对比较简单，变换结果的正确性和有效性也比较容易保证. 为支持这类变换系统的开发，我们提出一种“变换型语言”的概念，这种语言提供有力的定义和扩充语言的能力，以使用户能用它定义自己需要的规范描述语言；这种语言还支持程序变换的思想，允许用户在程序中自己定义变换的规则并控制变换的实现.

本文重点讨论变换型语言中变换的定义和控制问题，为叙述方便，变换的例子都以强类型的过程语言(如，Pascal)风格来书写；变换的效果主要以抽象数据类型的部分实现为目标. 至于变换型语言的扩充能力和完整定义将另有专文讨论.

§2. 变换的定义和控制

在变换型语言中，变换的定义用“变换模块”来表示，而变换的控制通过“变换控制命令”的执行来完成.

2.1 变换模块

一个变换模块给出一组相关的变换规则，这些规则定义了一个抽象数据类型的一种具体表示形式和这个抽象数据类型部分运算的实现细节.

变换模块的一般形式为：

```
TRANSFORM      变换模块名[(参数表)]                    /* 首部 */
DECLARATION                                            /* 说明部分 */
  REPRESENT    抽象变量说明    USING    表示变量说明
RULE                                                   /* 规则部分 */
  {REPRESENT   抽象表达式   USING   表示表达式
  |REWRITE     抽象表达式   USING   重写表达式
  |REPLACE     抽象语句     USING   置换语句   }
END
```

其中[，]，{，}，和 | 为元语言符号.

(1)首部

“变换模块名”是一个标识符，用它来标识一个变换模块.“(参数表)”是一个任选项，在无参数的情况下“( )”也一块略去.“参数表”由一串参数说明组成，并用逗号将它们隔开，每个参数说明给出一个参数的名字和类型.

(2)说明部分

这部分定义了本变换模块中使用的抽象变量，抽象类型，表示变量，表示类型以及抽象变量和表示变量之间的联系.

例如，在一个名为"Bool－Int"的变换模块中，可用以下的说明：

```
REPRESENT    b:bool    USING    j:int
```

表示本变换是用int(表示类型)来实现bool(抽象类型),本模块中出现的b是抽象变量(布尔型),而j则是b的表示变量(整型).这里抽象变量和表示变量之间呈现简单的一对一的关系.在一般情况下,也可以定义多对多的关系.例如,在考虑栈的顺序表示时,可以用一对多的变换,说明为:

```
REPRESENT    s : stack OF node
  USING      e : ARRAY 1…n0 OF node;
             t : 0…n0
```

也可以采用一对一的变换,说明为:

```
REPRESENT    s : stack OF node
  USING      a :RECORD
                 e : ARRAY 1…n0 OF node;
                 t : 0…n0
               END
```

这时抽象变量s:stack OF node用一个记录类型的具体变量a来表示,e和t作为记录的两个分量(或运算符)出现.

注意:这里的node和n0均可作为变换模块的参数出现,所不同的是node表示一个类型值,而n0表示一个整数值.

(3)规则部分

变换的规则分为三类,分别称为表示(REPRESENT)规则,重写(REWRITE)规则和置换(REPLACE)规则.前两类用于表达式的变换,而后一类用于语句的变换.

(a)表示规则的形式为:

```
REPRESENT    抽象表达式    USING    表示表达式
```

其中"抽象表达式"代表一类表达式的模式,它由抽象变量,参数,非终结符和抽象类型的运算符构成;"表示表达式"也是一类表达式的模式,它由表示变量,非终结符和表示类型的运算符构成;在表示规则中,抽象表达式的类型必须是本模块定义的抽象类型,而表示表达式的类型必须是本模块定义的表示类型.每个表示规则定义了一类抽象类型的表达式在该变换下的具体表示形式.

例如,在Bool—Int中可定义以下规则:

```
REPRESENT    b1 AND b2    USING    j1 * j2
```

此规则指出当布尔变量b1和b2的整数表示j1和j2已知时,布尔表达式b1 AND b2的表示可以通过j1 * j2得到.

注意:在变换规则中的抽象变量一般可以与任何具体表示已知的抽象类型的表达式匹配;用表示规则能得到的是与抽象表达式匹配的源程序中表达式的正确表示形式,而不是直接可用作重写的等价表达式.

(b)重写规则的形式为:

```
REWRITE    抽象表达式    USING    重写表达式
```

重写表达式的语法结构与表示表达式相同,但是在重写规则中,抽象表达式与重写表达式必须具有相同的类型.每个重写规则定义了一类表达式的等价的重写.

例如,在Bool—Int中可定义以下规则:

```
REWRITE    b    USING    j>0
```

(c)置换规则的形式为:

```
REPLACE　　抽象语句　　USING　　置换语句
```

其中“抽象语句”的构成类似“抽象表达式”;“置换语句”的构成类似“重写表达式”;其差别仅仅在于前者是一个语句模式,而后者是一个表达式模式.

例如,在Bool—Int中可以定义以下规则:

```
REPLACE　　b1:=b2　　USING　　j1:=j2
```

注意:作为赋值语句左部的抽象变量b1,只能与一个具体表示已知的抽象类型的变量匹配,而不能与抽象类型的一般表达式匹配.

2.2 变换控制命令

变换的控制是通过将“变换实例”作用于要变换的变量来完成的.变换实例可以是一个简单的变换模块名(无参的情况);或者(在有参的情况下)是一个变换模块名后缀以适当的实参,这时变换所使用的是用实参进行实例化以后的定义,它的抽象类型,表示类型和具体规则都可能依赖于实参.

根据变换控制命令出现的位置和形式,将它分成两类:局部控制和全局控制.

(a)局部控制

这种方式是将“/变换实例”后缀在被变换的变量的定义性说明处.

(b)全局控制

这种方式是把控制集中在一个特定文件中,每个控制命令的形式为:

```
APPLY　　变换实例　　TO　　变量扩展名
```

由于局部变量可以重名,被变换的变量可以前缀以该变量定义性出现所在的块名,从而构成唯一的扩展名.

用一个变换实例M去变换一个变量x时,首先将x用一个表示类型的变量来表示,将x的定义性说明改写成它的表示变量的定义性说明;然后将源程序中所有与x有关的语句和表达式,都尽可能用M中定义的规则进行变换.

2.3 例子

下面给出一个简单的变换例子.(2—1)给出了名为Bool_Int的完整变换模块,其中,第五行的注解刻划了表示变量与抽象变量之间的内在联系,称为“表示不变式”,它是程序变换正确性的主要依据;(2—2)给出了一个待变换的程序片断,变换的控制已经按局部方式插在源程序之中;(2—3)给出了这一变换的结果.

(2—1)一个简单的变换模块

```
    /* 用非负整数表示布尔量 */
TRANSFORM Bool_Int
DECLARATION
  REPRESENT b:bool              USING j: int
/* represent invariant: b=j>0 */
RULE
  REPRESENT  false              USING  0
  REPRESENT  true               USING  1
  REPRESENT  b1 OR b1           USING  j1 + j2
  REPRESENT  b1 AND b2          USING  j1 * j2
  REPRESENT  NOT b              USING  IF j > 0 THEN 0 ELSE 1
  REWRITE    b                  USING  j > 0
```

```
  REPLACE    b1 := b2        USING  j1 := j2
  REPRESENT  exp_x : bool    USING  IF x THEN 1 ELSE 0
END
(2-2)  VAR   b/Bool_Int, c/Bool_Int, d/Bool_Int : bool;
             ...
             c := false;
             ...
             IF (b OR c) AND (b OR d) THEN ···
             ...
(2-3)  VAR   b_j,c_j,d_j: int;
             ...
             c_j := 0;
             ...
             IF (b_j + c_j) * (b_j + d_j) > 0 THEN ···
             ...
```

§3. 若干问题的讨论

3.1 与传统模块的区别

变换模块与传统的模块(例如,Ada 中的 package)都体现了数据抽象的思想,然而过程型语言中的模块是传统的过程抽象的升华,主要采用了封装技术和信息隐蔽法则;[10]而变换模块为程序变换提供了定义的能力,它结合了函数型语言的基本思想和语义的形式描述技术.[2,4]在具体实现方法上自然也是完全不同的.

另外在模块的使用方式上,变换型语言提供了更为灵活的控制能力.在使用这种语言时,抽象类型、表示类型和抽象类型上运算的实现算法均具有相对独立性,即一个程序中同一类型的不同变量,允许采用不同的表示,或者虽然表示相同,但可采用不同的算法来实现有关的运算.[5,13,14]在 Ada 语言中,虽然这种独立性也存在,但一经确定,所有同一类型的变量及其运算,就只能采用统一的表示和实现方法.变换型语言的这种灵活性所产生的新问题是,必须考虑同一类型的不同表示之间的强制转换问题.

3.2 变换型程序的正确性

如果源程序正确性的已经保证,那么经过一次变换所得到的结果程序的正确性,可以通过变换规则的正确性和规则使用的正确性来保证.

(1)变换规则的正确性

(a)表示规则的正确性

以 REPRESENT　b1 AND b2　USING　j1 * j2为例,其正确性只要证明:

i) 抽象表达式 b1 AND b2的类型是抽象类型;(显然成立)

ii) 表示表达式 j1 * j2的类型是表示类型;(显然成立)

iii) 表示不变式对抽象表达式和表示表达式成立,即当 b1=(j1>0)和 b2=(j2>0)成立时,b1 AND b2=(j1 * j2)>0亦成立.(证明略)

(b)重写规则的正确性

以 REWRITE　b　USING　j>0为例,其正确性只要证明:

i) 抽象表达式 b 与重写表达式 j>0具有相同类型;(显然成立)

ii) 抽象表达式与重写表达式具有相同的值,即 b 与 j>0的值相等.(根据表示不变式,这也是显然的)

(c)置换规则的正确性

以 REPLACE b1:=b2 USING j1:=j2为例,其正确性只要证明同时执行这两个语句,表示不变式保持成立.采用最弱前置谓词 WP 的记法,[4,6]即要证明:WP("b1,j1:=b2,j2",b1=(j1>0))为真.(证明略)

(2)规则使用的正确性

一般来说,只要正确地通过匹配算法,将源程序中的一个语句或表达式与某规则中左部的模式成功地匹配,然后用匹配所得到的信息将规则右部模式中的标识符进行重写或替换,便得到源程序中这个表达式的一个具体表示(当该规则为表示规则);或者这个表达式的一个等价表达式(当该规则为重写规则);或者得到源程序中这个语句的一个等价语句(当该规则为置换规则).对第一种情况所得到的表示表达式可作为进一步匹配和重写的前提;而后两种情况则可以立即用等价的表达式(或语句)去重写(或置换)源程序中的相应部分.

需要注意的是,有些规则的定义和使用是有条件的,我们将这些条件称为规则的前置条件,在使用这类规则前,必须首先检验各自的前置条件.例如,在使用一个 k 位布尔向量来实现其元素值小于 k 的正整数集合时,可用下面的规则定义插入运算:

```
REPLACE    s:=sU{i}    USING    b.i:=true
```

本规则使用时,必须保证0≤i≤k−1.

3.3 关于变换表示的一般性

本文所引入的变换机制,除了可以用来表示从抽象数据类型到具体类型的变换外,也可以用来表示从不确定程序到确定程序的转换,对谓词变量的换名,以及一般的坐标变换.[4,7]限于篇幅,这里不再详述.

结束语:变换型语言的研究在国外也刚刚起步,美国 Cornell 大学 D. Gries 教授领导的研究组,从1988年开始研究和开发的"polya"语言是第一个在高级语言中引入变换功能的语言.* 该项研究得到了美国国家基金会的资助.笔者从该研究组成立初起,便加入了该组的工作,回国后又继续这方面的研究,设计并实现了一个程序变换的模拟系统.关于变换型语言的全貌,特别是关于语言的扩展功能也在进一步研究之中.

参考文献

1 A. V. Aho, J. E. Hopcroft and J. D. Ullman, Data Structures and Algorithms, Addision-Wesley, Reading, 1985.

2 J. Backus, Can Programming be Liberated from the VonNeumann Style?, Comm. ACM, August, 1978.

3 W. Chen and J. T. Udding, Towards a Calculus of Data Refinement, LNCS 375, Spring Verlag, NY 1989.

4 E. W. Dijkstra, A Discipline of Programming, Prentice Hall, Englewood Cliffs, New Jersey, 1976.

5 D. Gries and J. Prins, A New Notion of Encapsulation, Proc. SIGPLAN 85 Sympsium on Language, Issues in Programming Environments, 1985.

6 D. Gries, The Science of Programming, Spring-verlag, 1981.

* polya 语言的文本尚未正式发表.

7 D. Gries and D. Volpano, The Transform——A New Language Construct, Tech. Rpt. CS Dept. Cornell Univ. 1989.

8 C. A. R. Hoare, Proof of Correctness of Data Representations, Acta Informatica, 1972.

9 J. E. Hopcroft and K. W. Kennedy, Computer Science——Achievements and Opportunities, Society for Inductrial and Applied, Math. 1989.

10 E. Horowitz, 程序设计语言基础（中译本），北京大学出版社（1983/1990）.

11 D. E. Knuth, The Art of Programming, Vol I, Fundamental Algorithms, Addison-Wesley, Reading, 1963.

12 J. M. Morris, The Laws of Data Refinement, Acta Informatica 26, 1989.

13 J. Prins, Partial Implementations in Program Derivation, Ph. d. Thesis, CS Dept. Cornell Univ. 1987.

14 D. Volpano, Towards a Notion of Module for Data Abstraction, Tech. Rpt. CS Dept. Cornell Univ. 1989.

15 仲萃豪，冯玉琳，陈友君，程序设计方法学，北京科学技术出版社，1985.

16 张乃孝等编译，美国计算机研究报告，模式识别与人工智能，Vol. 3, No. 4, 1990. 12.