# 传统程序设计语言的面向对象扩充的若干技术*

李宣东 郑国梁

(南京大学计算机科学与技术系 南京 210093)

**摘要** 本文论述用转换方式实现传统程序设计语言的面向对象扩充,给出了一组关键技术以解决由子类型多态和动态定连所引出的问题,其基本思想是将类和对象类型分开处理,从而避免转换过程中的重复工作.这些技术具有适用性强、功效高的特点,并且在PASCAL语言的面向对象扩充NDOOP和MODULA-2语言的面向对象扩充NDOOM的实现中被采用,效果令人满意.

**关键词** 面向对象程序设计,转换方式,动态定连.

目前现有的面向对象程序设计语言的实现方式有2种:解释实现(如Smalltalk[1])和编译实现(如Eiffel[2]).而编译实现又可分为直接实现和转换实现2类:直接实现是指直接构造语言的编译系统,如语言Eiffel,C++[3]等;转换实现是指将要实现的面向对象语言编写的程序转换为某种传统语言编写的程序,用传统语言的实现系统实现它,如语言DRAGON[4],C++的早期版本等.一般认为,各种实现方式各有其优缺点:解释实现灵活性大,适于支持原型开发,但功效不高;编译实现功效较高,但不适于支持原型开发;直接实现功效高,但工作量大;转换实现有一定功效,工作量小.在语言设计的初级阶段,可用转换实现方式,以最小代价获得语言原型,不断修改和推广;在语言设计和使用均已成熟的情况下,可采用直接实现方式,提高语言的功效以使其更具生命力.

目前已有的面向对象程序设计语言可以分为2大类:一类是纯面向对象程序设计语言,如Smalltalk-80,Eiffle等;另一类则是在传统的过程式语言或其它语言中加入面向对象机制所形成的传统语言的面向对象扩充,其侧重点在于提高速度,特别是易于让已熟悉传统语言的软件开发人员尽快接受和掌握面向对象程序设计技术.由于历史软件积累和语言的使用惯性以及其它各方面的原因,传统程序设计语言目前和今后一个时期仍然是软件市场上的主导开发语言之一.因此,对传统程序设计语言进行面向对象扩充已经成为程序设计语言的发展趋势之一.

传统程序设计语言的面向对象扩充经过一段时期的发展之后,在语言的设计方面更加

---

* 本文研究得到国家教委博士点基金和江苏省科技应用基础基金资助.
作者李宣东,1963年生,博士,讲师,主要研究领域为面向对象程序设计语言,软件工程.郑国梁,1937年生,教授,博士导师,主要研究领域为软件工程.

本文通讯联系人:李宣东,南京210093,南京大学计算机科学与技术系

本文1995-12-13收到修改稿

注重对面向对象程序设计的充分支持,语言一般都具有对象、类、对象类型、继承、子类型多态和动态定连等概念和机制;在语言的实现方面更加注重功效.这样,就给语言的实现工作,特别是用转换方式实现传统程序设计语言的面向对象扩充带来一定的困难.

本文论述用转换方式实现传统程序设计语言的面向对象扩充,给出了一组关键技术.这些技术具有适用性强、功效高的特点,和国外类似的工作[4~6]相比具有一定的优势,并且在PASCAL语言的面向对象扩充NDOOP[7]和MODULA-2语言的面向对象扩充NDOOM[8]的实现中被采用,效果令人满意.本文下面将传统程序设计语言简称为GPL,传统程序设计语言的面向对象扩充简称为OOGPL.

## 1　用转换方式实现传统程序设计语言的面向对象扩充

根据OOGPL本身的特点,其实现可采用转换实现方式.设OOGPL是GPL的面向对象扩充,则将GPL称为OOGPL的基语言,GPL和OOGPL的非基语言GPL′均称为OOGPL的实现语言.这样,采用转换方式实现OOGPL又可以分为2种类型:

(1)用OOGPL的基语言GPL实现OOGPL;

(2)用OOGPL的非基语言GPL′实现OOGPL.

用OOGPL的基语言GPL实现OOGPL的优点在于工作量小,但对某些OOGPL来说,由于其基语言GPL的描述能力有限,因而用GPL实现OOGPL的难度较大,且功效降低,因而需要用非基语言GPL′实现它.

一般来说,实现语言需支持记录类型和动态数据结构(指针)方能比较方便和有效地实现OOGPL.

### 1.1　实现系统的结构

一个OOGPL实现系统的结构如图1所示.图1所示转换系统是OOGPL实现系统的核心,其把用OOGPL书写的程序转换为用GPL(GPL′)书写的程序,因而是本文讨论的重点.


图1　OOGPL实现系统

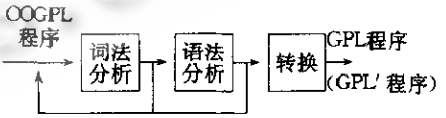


图2

### 1.2　转换系统

转换系统由词法分析、语法分析和转换3部分组成,见图2.

(1)词法分析

词法分析部分的功能是对构成OOGPL程序的字符串进行扫描和分解,识别出单词,检查词法错误.

(2)语法分析

语法分析的主要功能是分析OOGPL中相对于GPL(GPL′)新增加的语法结构(主要包括对象、对象类型、类、继承和消息发送等语法结构的分析)、检查语法错误、生成转换所需的一些数据和表格,为转换做准备工作.

(3)转换

转换部分的工作是实现OOGPL程序到GPL(GPL′)程序的转换.由于OOGPL程序与GPL(GPL′)程序在结构上类似,因而转换工作的主要重点是用GPL(GPL′)中的机制实现OOGPL中新增加的机制:对象、对象类型和类.转换工作面临的主要困难是:

(1)GPL(GPL′)是静态强类型的,无法直接实现OOGPL中的子类型多态;

(2)GPL(GPL′)是静态定连的,无法直接实现OOGPL中支持的动态定连.

克服以上2方面的困难需要采用下节介绍的用转换方式实现传统程序设计语言的面向对象扩充的若干技术.

## 2 用转换方式实现传统程序设计语言的面向对象扩充的若干技术

将OOGPL程序转换为GPL(GPL′)程序,关键在于如何用GPL(GPL′)中的机制实现OOGPL中的对象、对象类型和类.这个问题可通过介绍以下几个关键转换技术加以回答.

### 2.1 类的转换

在OOGPL中,对象封装了数据和操作.类是创建对象的模板,或者是实现对象的机制,由实例变量集和方法集构成.由于GPL(GPL′)中没有直接支持封装数据和操作的机制,OOGPL中的对象的状态和操作在GPL(GPL′)只能分别用变量和作用于变量上的过程(函数)表示.因此,OOGPL中的类的实例变量集和方法集在GPL(GPL′)程序中分别用记录类型和一组过程(函数)表示,也即OOGPL程序中的一个类转换为GPL(GPL′)程序中的一个记录类型和一组过程(函数).

设在OOGPL程序中有类$A$,其实例变量集为$\{v_1:T_1,v_2:T_2,\ldots,v_n:T_n\}$,方法集为$\{m_1,m_2,\ldots,m_n\}$,则类$A$的实例变量集转换为GPL(GPL′)程序中的记录类型$RC_A$:

```
RC_A=RECORD
        v1:T1;
        v2:T2;
        ……;
        vn:Tn;
     END
```

记录类型$RC_A$称为类$A$的转换类型;类$A$的方法集转换为GPL(GPL′)程序中的一组过程(函数):

| 方法 | 过程(函数) |
|---|---|
| $m_1(P_1)$ | $RC\_A\_m_1(V,P_1)$ |
| $m_2(P_2)$ | $RC\_A\_m_2(V,P_2)$ |
| …… | …… |
| $m_n(P_n)$ | $RC\_A\_m_n(V,P_n)$ |

其中$P_i=p_{i1}:T_{i1},p_{i2}:T_{i2},\ldots,p_{in}:T_{in}$,$V=v_1:T_1,v_2:T_2,\ldots,v_n:T_n$.过程(函数)$RC\_A\_m_i$ $(i=1,2,\ldots,n)$称为类$A$的转换过程(函数).

### 2.2 对象类型的转换

在OOGPL程序中,对象由对象类型说明,由指引元变量间接标识.对象类型说明了一组表示对象行为的方法,类则实现了这组方法.在面向对象程序设计语言中,对象类型和类可以相互独立,也可以合二为一.为了避免转换过程中的重复工作,我们将类和对象类型分开处理,即使在语言中类和对象类型合二为一,相应的转换过程仍然分为2步:类的转换和对

象类型的转换.不失一般性,这里假定在OOGPL程序中标识对象的指引元变量按如下形式说明:

$$i:T_i \quad \text{IMPLEMENTED BY} \quad C_i,$$

$T_i$是指引元变量$i$所标识的对象的类型;$C_i$是实现指引元变量$i$所标识的对象的类.

由于OOGPL程序中的类在GPL(GPL′)程序中由转换类型和转换过程(函数)表示,因此似乎OOGPL程序中的对象在GPL(GPL′)程序中可以由具有转换类型的变量表示.但是,由于OOGPL允许子类型多态,因而在GPL(GPL′)程序中这样表示对象将给转换工作带来困难.现设在OOGPL程序中有如下对象说明:

$$i:T_i \quad \text{IMPLEMENTED BY} \quad C_i;$$

$$j:T_j \quad \text{IMPLEMENTED BY} \quad C_j;$$

且由变量$i$和$j$所标识的对象在GPL(GPL′)程序中分别用类$C_i$和类$C_j$的转换类型$RC_C_i$和$RC_C_j$的变量$i'$和$j'$表示:　　$i':RC_i;$　　$j':RC_j;$

若在OOGPL程序中,有$T_i<T_j$,则赋值语句$i:=j$和类似的形、实参数匹配是合法的.但是,在GPL(GPL′)程序中,相应的赋值语句$i':=j'$和类似的形、实参数匹配只有在变量$i'$和$j'$的类型相同时才合法(这里$i'$和$j'$的类型$RC_C_i$和$RC_C_j$均为记录类型).这就是转换过程中由于OOGPL支持子类型多态而产生的问题.

要解决上述问题,必须要求OOGPL程序中具有子类型关系的对象类型的实例对象在GPL(GPL′)程序中用同一记录类型的变量表示,称这个记录类型为对象类型的转换类型.

构造OOGPL程序中的对象类型的转换类型可按如下步骤进行:

(1)将OOGPL程序中的所有对象类型按子类型关系划分等价类,形成一组对象类型集合:

$$ST_1,ST_2,\ldots,ST_n, ST_i \cap ST_j = \oint \quad (i\neq j, i,j=1,2,\ldots,n);$$

(2)根据对象类型集合$ST_1,ST_2,\ldots,ST_n$构造一组类集:

$$CST_1,CST_2,\ldots,CST_n, CST_i=\{C \mid \exists\ t\in ST_i \cdot C \text{ 是 } t \text{ 的实现}\} \quad (i=1,2,\ldots,n);$$

(3)设$CST_i=\{C_{i1},C_{i2},\ldots,C_{in}\}$,类$C_{ij}$的转换类型为$RC_C_{ij}$　$(j=1,2,\ldots,n)$,则记录类型$R_i$即是对象类型$t\in ST_i$的转换记录类型:

```
R_i=RECORD
      LRC_i1:RC_C_i1
      LRC_i2:RC_C_i2
      ......
      LRC_in:RC_C_in
    END
i = 1, 2, ..., n;
```

(4)显然,用具有记录类型$R_i$的变量来表示对象类型$t\in ST_i$的实例对象可能造成大量的空间浪费.因此,将记录类型$R_i$修改如下:

```
R_i=RECORD
      LRC_i1:POINT TO RC_C_i1;
      LRC_i2:POINT TO RC_C_i2;
      ......
      LRC_in:POINT TO RC_C_in;
    END
```

这样,OOGPL程序中的变量$i$所标识的对象(其类型$t\in ST_i$,类$C_{ik}\in ST_i$)在GPL(GPL′)程序中用变量$i':R_i$表示,且有$i'.LRC_{ik}\neq NIL, i'.LRC_{ij}=NIL$ $(j\neq k, j=1,2,\ldots,$

$n$),从而避免了空间浪费.

解决由于OOGPL支持子类型多态而导致的转换问题,还可以采用变体记录类型的方法[4],但是必须要求GPL(GPL′)支持变体记录类型,相比之下这里提出的方法适用性更强.

### 2.3 动态定连的实现

在OOGPL中,对象类型说明了其实例对象的一组方法,这组方法由类实现.设在OOGPL程序中有如下对象说明:$i:T$ IMPLEMENTED BY $C$,则向变量$i$所标识的对象发送消息$m(i.m)$的含义是调用类$C$中实现$m$的方法$n$($n$和$m$可以不同名,但参数和结果类型必须相同).因此,OOGPL程序中的消息发送$i.m$似乎在GPL(GPL′)程序中可以转换为调用类$C$的转换过程(函数)$RC\_C\_n$.但是,由于OOGPL支持子类型多态,因此无法静态判定变量$i$所标识的对象的对象类型和类.从而要求在GPL(GPL′)程序中实现过程(函数)的动态定连.

由于GPL(GPL′)是静态定连的,不直接支持过程(函数)的动态定连.因此,为在GPL(GPL′)程序中实现动态定连,需要为OOGPL程序中每个对象类型所具有的每个方法都建立一个相应的过程(函数),这些过程(函数)称为对象类型的转换过程(函数):

| 对象类型$T$中的方法 | 对象类型$T$的转换函数 |
|---|---|
| $m_1(P_1)$ | $RT\_T\_m_1(r:RT\_T,P_1)$ |
| $m_2(P_2)$ | $RT\_T\_m_2(r:RT\_T,P_2)$ |
| …… | …… |
| $m_n(P_n)$ | $RT\_T\_mn(r:RT\_T,P_n)$ |

其中$P_i=P_{i1}:T_{i1},P_{i2}:T_{i2},\ldots,P_{in}:T_{in}(i=1,2,\ldots,n)$,$RT_T$是$T$的转换类型.

对象类型$T$的转换过程(函数)的功能是在GPL(GPL′)程序中实现过程动态定连.设在OOGPL程序中有如下对象说明:$i:T$ IMPLEMENTED BY $C$,对象类型$T$的转换类型为$RT_T$,变量$i$所标识的对象在GPL(GPL′)程序中用变量$i':RT_T$表示,则OOGPL程序中向$i$标识的对象发送消息$m(i.m(p_1))$转换为GPL程序中的过程调用:$RT\_T\_m(i',p_1)$.

在构造对象类型的转换过程(函数)之前,首先要在对象类型的转换类型(记录类型)中增加2个域:Typename和Classname,

```
R=RECORD
    LRC1:POINT TO RC_C1;
    LRC2:POINT TO RC_C2;
    ……
    LRCn:POINT TO RC_Cn;
    Typename : STRING;
    Classname: STRING
  END
```

域Typename存放对象的类型名,域Classname存放对象的类名.这样在GPL程序中,变量$i$:$R$所表示的对象的类型名为$i.Typename$,类名为$i.Classname$.

为了完成过程的动态定连功能,对象类型$T$的转换过程(函数)$RT_T_m_i(r:RT_T,P_i)(i=1,2,\ldots,n;RT_T$为$T$的转换类型)的构造可按以下步骤进行:

(1)构造对象类型$T$及其子类型构成的集合$TS$:$TS=\{t|(t=T)\vee(t<T)\}$;

(2)根据$TS$,构造类集$TC$:$TC=\{C|\exists\ t\in TS\cdot C$实现$t\}$;

(3)根据$TS$,$TC$,构造集合$TSC$.集合$TSC$的元素是个二元组$(t,c)$:$(t\in TS)\wedge(c$

$\in TC) \wedge (c$ 实现 $t)$;

(4)设 $TSC=\{(t_1,c_2),(t_2,c_2),\ldots,(t_n,c_n)\}$;$T$ 的转换类型为 $RT\_T$:

```
RT_T=RECORD
        LRC1:POINT TO RC_C1;
        LRC2:POINT TO RC_C2;
        ......
        LRCn:POINT TO RC_Cn;
        LRCn+1:POINT TO RC_Cn+1;
        ......
        LRCm:POINT TO RC_Cm;
     END
```

对任一 $(t_j,c_j)\in TSC$,$t_j$ 是 $T$ 或 $t_j$ 是 $T$ 的子类型;$t_j$ 中对应于 $T$ 中的方法 $m_i$ 的方法为 $mt_{ji}$;类型 $t_j$ 中的方法 $mt_{j1},mt_{j2},\ldots,mt_{jn}$ 分别对应于类 $c_j$ 中的方法 $mc_{j1},mc_{j2},\ldots,mc_{jn}$;类 $c_j$ 的转换类型为 $RC\_c_j$:

```
RC_cj=RECORD
        vj1:Tj1
        vj2:Tj2
        ......
        vjn:Tjn
      END
```

类 $C_j$ 的转换过程(函数)为:

$RC\_c_j\_mc_{jk}(v_{j1}:T_{j1},v_{j2}:T_{j2},\ldots,v_{jn}:T_{jn},P_t),K=1,2,\ldots,n;j=1,2,\ldots,n,$

则对象类型 $T$ 的转换过程(函数)$RT\_T\_m_i(r:RT\_T,P_t)$ 的体是:

```
BEGIN
  CASE r.Typename OF
    "t1":IF r.classname="c1"
         THEN RC_c1_mc1i(r.LRC1↑.v11:T11, r.LRC1↑.v12:T12,..., r.LRC1↑.v1n:T1n,Pi);
    "t2":IF r.classname="c2"
         THEN RC_c2_mc2i(r.LRC2↑.v21:T21,r.LRC2↑.v22:T22,..., r.LRC2↑.v2n:T2n,Pi);
    ......
    "tn":IF r.classname="cn"
         THEN RC_cn_mcm(r.LRCn↑.vn1:Tn1,r.LRCn↑.vn2:Tn2,..., r.LRCn↑.Vmn:Tnn,Pi);
    END
  END
```

显然,若在已有系统中加入新的对象类型(类),则已有对象类型的转换类型和转换过程可能需要重新生成,但对已有类的转换工作不需重复,这也是我们将类和对象类型分开处理的用意所在.

### 2.4 继承的实现

由于 OOGPL 允许类间的继承,因而就某个具体的类来说其拥有的一些方法是从父类中继承来的,在该类的实现部分没有定义.就某个类 $A$ 的某个具体转换过程 $RC\_A\_m$ 来说,类 $A$ 中的方法 $m$ 在 $A$ 的实现部分有定义,则可用方法 $m$ 的体作为 $RC\_A\_m$ 的体来定义 $RC\_A\_m$;若类 $A$ 中的方法 $m$ 是从 $A$ 的父类继承来的,且方法 $m$ 在 $A$ 的实现部分无重定义,则无法直接定义 $RC\_A\_m$.在转换 OOGPL 程序为 GPL(GPL′)程序的过程中,继承的实现就是给出对应于类中继承方法的转换过程(函数)的定义.以下将继承分类为静态继承和动态继承,分别讨论相应的实现.

(1)静态继承的实现

静态继承的实现较为简单.静态继承的操作语义可以通过类似于Smalltalk－80中的方法查询算法给出:

消息接收者接到消息后,在自己所属类中查询与消息匹配的方法.若找不到,则继续在超类中查找;若再找不到,则继续在超类的超类中查找.如此一直进行下去,直到找到为止.

若消息的接收者为*self*,则查询从包含*self*的方法所在的类开始.

若消息的接收者为*super*,则查询从包含*super*的方法所在的类的直接超类开始.

根据静态继承的操作语义不难确定类$A$中继承的方法$m$的实现代码所在的类,设其为$B$,且类$A$中的方法$m$对应于类$B$中的方法$n$($m$和$n$可以不同名,但参数和结果类型必须相同),则相应于方法$m$的类$A$的转换过(函数)$RC_A_m$可用相应于方法$n$的类$B$的转换过程(函数)$RC_A_n$加以定义.

设类$B$的实例变量集为$(vB_1:T_1,vB_2:T_2,\ldots,vB_m:T_m)$,类$A$的实例变量集为$(vA_1:T_1,vA_2:T_2,\ldots,vA_m:T_m,\ldots,vA_n:T_n)$,则过程(函数)$RC_A_m(vA_1:T_1,vA_2:T_2,\ldots,vA_n:T_n,P)$的体为:

```
BEGIN
  RC_B_n(vA1,vA2,...,vAm,P)
END
```

其中$P=p_1:TP_1,p_2:TP_2,\ldots,p_n:TP_n$.

(2)动态继承的实现

动态继承的实现相对复杂,动态继承的操作语义一般由Smalltalk-80中的方法查询算法(1)给出:

消息接收者接到消息后,在自己所属类中查询与消息匹配的方法.若找不到,则继续在超类中查找;若再找不到,则继续在超类的超类中查找.如此一直进行下去,直至找到为止.

若消息的接收者为*self*,则查询从发送者所属的类开始.

若消息的接收者为*super*,则查询从包含*super*的方法所在的类的直接超类开始进行.

实现动态继承首先要给出类中每个方法的调用树.类$A$中方法$m$的调用树回答了这样一个问题:"如果向类$A$的实例对象发送消息$m$,将会唤醒(调用)类$A$,类$A$的父类以及类$A$的祖先中的哪些方法?"

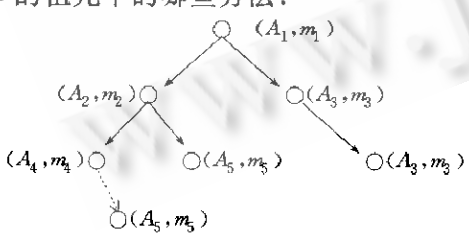


图3 类$A$中方法$m$的调用树

例如,图3是类$A$中方法$m$的调用树,给出的类$A$中方法$m$的调用树的含义是:若向类$A$的实例对象发送消息$m$,首先调用类$A_1$中的方法$m_1$;在方法$m_1$的执行过程中将要调用类$A_2$中的方法$m_2$和类$A_3$中的方法$m_3$;在方法$m_2$的执行过程中将要调用类$A_4$中的方法$m_4$和类$A_5$中的方法$m_5$;在方法$m_3$的执行过程中将要调用类$A_6$中的方法$m_6$;由于方法$m_4$的执行过程中又要调用类$A_2$中的方法$m_2$,出现了递归调用,因此调用树的生成在节点$(A_4,m_4)$终止.

一个方法调用树产生后,可以根据以下规则进行剪枝:

对调用树中除根以外的任一结点,设其标志类$A_n$中的方法$m_n$,若在调用树中以该结点为根所形成的树与类$A_n$中方法$m_n$的调用树相同,则将该结点的所有后继(子孙)结点剪

去,使该结点成为调用树中的叶子.

一个经过剪枝的方法调用树的结点分为3种类型:第1种称为非终结点,是指调用树中除叶子以外的结点,如上图中的$(A_1,m_1)$,$(A_2,m_2)$,$(A_3,m_3)$;第2种称为虚终结点,这类结点是调用树中的叶子,但是这些结点中的方法还要调用其它类中的方法,形成递归调用,如上图中的$(A_4,m_4)$;第3种结点称为终结点,这类结点是调用树中叶子,但不是虚终结点,如上图中的$(A_5,m_5)$,$(A_6,m_6)$.

由方法调用树可以看出,动态继承可用方法查询方法实现,但这样效率太低.我们采用将动态继承化为静态继承的方法实现动态继承:复制方法调用树中的非终结点和虚终结点中指出的方法的实现代码以形成类中的新方法.显然,这样实现动态继承将产生代码冗余,但可以提高程序的执行速度,而对方法调用树进行剪枝也正是为了将代码冗余减少到最低程度.

## 3 小　结

本文比较全面地论述了用转换方式实现传统程序设计语言的面向对象扩充,给出了一组关键技术以解决转换实现中的难点:子类型多态和动态定连的实现.

文献[4]介绍了用转换方式实现的ADA语言的面向对象扩充DRAGON,为解决子类型多态所引出的问题,其采用了变体记录类型支持对象类型的转换;文献[8]介绍了用转换方式实现的Modula-2语言的面向对象扩充OOM,但是OOM不支持子类型多态;文献[7]以转换规则的形式描述了Objective TURING到TURING语言的转换,但是对子类型多态和继承的转换实现没有论述.因此,本文工作与上述国外类似的工作相比具有一定的优势.

## 参考文献


1 Goldberg A, Robson D. Smalltalk-80: the language and its implementation. Addison-Wesley, 1983.

2 Meyer B. Eiffel: the language. Prentice-Hall, 1992.

3 Stroutrup B. The C++ programming language. Addison-Wesley, Menlo Park, 1991.

4 Atkinson C. Object-oriented reuse. Concurrency and Distribution. Addison-Wesley, 1991.

5 Cordy R J, Promislow E. Specification and automatic prototype implementation of polymorphic objects in TURING using TXL dialect processor. IEEE Computer Language Conference, 1990.

6 Thomas R. A proposal for object-oriented modula-2. Second International Modula-2 Conference, 1991.

7 李宣东,郑国梁.一个Pascal的面向对象扩充的设计与实现.软件学报,1996,7(1):9~15.

8 李宣东,郑国梁.对Modula-2进行面向对象扩充.计算机学报,1996,19(1):30~35.


# IMPLEMENTING OBJECT-ORIENTED EXTENSIONS OF TRADITIONAL PROGRAMMING LANGUAGES


LI Xuandong　ZHENG Guoliang

(*Department of Computer Science　Nanjing University　Nanjing*　210093)

**Abstract** This paper describes a transformation approach to implementing object-oriented extensions of traditional programming languages (OOGPLs), which OOGPLs are implemented using a preprocessor that translates from the program in OOGPL to an equivalent one in a traditional programming language. Some techniques are given to overcome difficulties with allowing polymorphism and dynamic binding. By them the authors have implemented NDOOP which is an object-oriented extension of Pascal and NDOOM which is an object-oriented extension of Modula-2.

**Key words** Object-oriented programming language, transformation approach, dynamic binding.