面向对象程序设计体裁嵌入 FFP-AST 系统*

江明德 菊 燕

(成都电子科技大学计算机系)

OBJECT-ORIENTED PROGRAMMING PARADIGM EMBEDED IN FFP-AST SYSTEM

Jiang Mingde and Ju Yan

(*Computer Departmant, University of Electronic Science and Technology of Chengdu*)

ABSTRACT

Basing upon a strict mathematical theory, the programming language FFP[1] is a purely functional programming language and a subset of the reduction language $\mathcal{L}_4^{[2,3]}$. In this paper the object-oriented programming (OOP) paradigm is considerably succinctly embeded in the FFP-AST system. By this means, on the one hand, it is revealed that the object-oriented programming paradigm (OOPP) and the functional programming paradigm (FPP) are closely related. On the other hand, the semantics of OOPP described by the FFP-AST is settled. In essence, a programming language is proposed, being possessed of both FPP and OOPP. To put it in a nutshell, the methodology penetrating the work in this paper is the idea of closely integrating the automation with practice.

摘 要

FFP 语言[1]是这样的一种纯粹的泛函程序设计语言，它奠基于严格的数学理论基础之上，是归约语言 $\mathcal{L}_4$ 的一个子类[2,3]。本文将面向对象编程(OOP)体裁相当简洁地嵌入到FFP-AST 系统中。这样，一方面，揭示了面向对象编程体裁(OOPP)与泛函编程体裁(FPP)之间的近亲关系；另一方面，为OOPP 奠定了

* 1989 年7 月3 日收到，1990 年1 月7 日定稿，本课题得到国家高技术研究发展计划基金的资助。

FFP-AST语义描述。本文实质上也提供了一种“加于FFP语言之上的兼备FPP和OOPP的编程语言”。贯穿文中的方法论归结为: 紧密联系和使用自动机这一概念。

§1. 问题之提出

以Smalltalk-80为代表的OOP语言的问世, 将计算机软件的“虽互有联系但并不总是体现于一身的”重要的几个方面基本上统一成一个整体。OOP语言看来是这样的“五位一体”的东西: 一种编程语言, 一种操作系统, 一种编程环境, 一套概念设计和程序设计的方法学, 一种知识表示的工具——知识表示语言。OOP语言以其迅猛之势发展着, 博得了系统软件界及应用软件界各类人员的关注。

在AI语言中引入OOPP是合适的。为探索一种新AI语言, 弄清楚下述诸问题殊属必要: ①OOP语言与已存在的新一代(泛函的或逻辑的)编程语言的关系究竟如何? ②能否在现有的语义描述工具的基础上为OOP语言奠定一种简明的精确的语义描述? ③初步设计和实现一种简单的OOP语言(它还可用作某些“加糖衣使之易读的”OOP语言的中间语言), 此种语言自然地结合某种新一代编程体裁与OOP体裁。

提出的三个问题相互关联, 本文一揽子地予以解决。

§2. 方法论

OOPP显示着自动机网络的特质[4], 无疑地, 在逻辑的[4]或泛函的体裁框架内, 均有可能解决上节所提出的问题。本文以“扩展并具体化泛函和自动机这两概念”作为设计方法论, 来联系OOP实际。再者, 鉴于FFP语言是十分简单的而又奠基于严格数学理论基础之上的一种泛函编程语言, 并且它已予实现[5,6], 因此, 应用FFP作为描述工具和实现语言的“泛函·自动机·FFP-AST”三位一体的构思, 会使我们在理论上和实践上一揽子地而又易于受人理解地完成所提出的任务。

具体些说, 我们将把(类型)构造子、类、Ω对象(OOP中对象)均予以泛函对待, 消息传递、继承机制则予以泛函型调用对待。

§3. FFP-AST系统[1]

FFP-AST系统是一个自动机, 它由三部分构成(图1): ①作用式子系统, 它是FFP语言系统。②状态D, 它是作用式子系统的诸定义所形成的集。③转移法则集, 它描述输入是怎样被变换成输出的, 状态D又是怎样变化的。

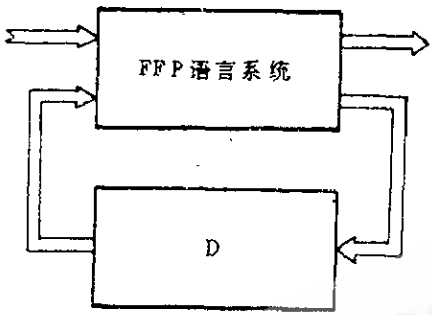

图 1　FFP-AST 系统作为自动机

§4. 构造子作为泛函

构造子在一定意义上带有函数的性质，是一种“非执行性的函数”。将构造子对待成泛函，显示了FFP的表达能力之强大。这样做并不违背Backus设计FFP的初衷，也不损害FFP之单一种数据结构(序列)的“概念均一，简明，便于数学论证”，因为：构造子(例如，记录作为泛函)既是泛函又是序列，象用户定义(一般的)泛函一样，这里泛函和序列均为FFP中的基本概念，再不需要任何其它“糖衣”。再者，FFP中的记录与通常语言中各种带糖衣的记录，形式上很为接近，易读性不致损失；何况，在相当大程度上易读性有一个先入为主的习惯问题。如果必要(如将FFP作为中间语言)，FFP的记录可极为自然地成为通常记录的译本。

虽然记录完全可在基于FFP-AST系统上构作起来的OOP语言里定义为一个类，但是，为了便于论述，我们在FFP语言中定义记录，它将作为一个泛函。一般地说，数据结构的构造子在FFP中均可定义成泛函，伴随以相应的一些函数。以RECORD为名的泛函所形成的泛函型采取形式<RECORD, <<lab1, exp1>, <lab2, exp2>, ..., <labn, expn>>>，它作用于任一非⊥对象上等于它自己。有四个函数与记录这一数据结构相伴：RecordSelect, RecordStore, RecordStore1, RecordWith。

Def RECORD≡not ∘ eq ∘ [2, % ⊥]→1; % ⊥ /*% 表示恒常泛函*/

Def RecordSelect≡/* RecordSelect: {<labi, record>} → ({expi}∪{⊥}). 从记录record中选出标记以labi的表达式expi；若无，给出⊥*/

and ∘ [pair, eq ∘ [1 ∘ 2, % ∘ RECORD]] → apply ∘ [[% FETCH, 1], apply ∘ [[% ALFA, [% BU, % apndl, % CELL]], tl]]; % ⊥

Def RecordStore≡/*RecordStore: {<<lab, exp>, record> } → ({newRecord} ∪ {⊥}). 将<lab, exp>存入record中作为一个新成分。若record中原来就有标记为lab的这一成分，则此原成分被删去*/and ∘ [pair ∘ 1, and ∘ [pair, eq ∘ [1 ∘ 2, % RECORD]]] → apndl ∘ [% RECORD, id]。

αtl ∘ apply ∘ [[% STORE, 1 ∘ 1], apply ∘ [[% ALFA, [%BU, % apndl, % CELL]], tl] ∘ 2]; % ⊥

Def RecordStore1 ≡ /* RecordStore1: {<<lab, exp>, record>} → ({newRecord} ∪ {⊥})。更换Record中标记为lab的那一成分以新成分<lab, exp>。若Record中原来就不存在标记为lab的成分，则给出⊥*/and ∘ [pair ∘ 1, and ∘ [pair, eq ∘ [1 ∘ 2, % RECORD]]] → apndl ∘ [% RECORD, id]。

αtl ∘ apply ∘ [[% STORE1, 1 ∘ 1], apply ∘ [[%ALFA, [% BU, % apndl, % CELL]], tl] ∘ 2]; % ⊥

Def <STORE1, n> ≡ pair → (eq ∘ [<pop, n> ∘ 2, 2] → % ⊥; <push, n> ∘ [1, <pop, n> ∘ 2]); % ⊥

/* 严格说来，应该定义STORE1, 而不是<STORE1, n>。这里为直观起见定义了后者。关于pop, push, FETCH(即↑), STORE (即↓) 诸泛函，参见[1]*/

Def RecordWith ≡ /* RecordWith:{record, <<lab11, exp11>, ···, <lab1m, exp1m>>>} → {<newRecord>}.新记录newRecord, 除去标记为lab11, ..., lab1m 的诸成分外，与原记录record 一样*/and ∘ [pair, eq ∘ [1 ∘ 1, % RECORD]] → 1 ∘ (while not ∘ eq ∘ [tl ∘ 2, % <>]

[RecordStore ∘ [1 ∘ 2, 1], tl ∘ 2]); % ⊥

§5. 嵌入于FFP-AST系统中的OOP语言的总体结构

OOP语言的总体结构可看成是一架自动机(图2), 它由下述诸部分组成: (i) 记忆装置: 存放了各对象(类的实例)的诸实例变元的值组。每一类的诸类变元的值组也均存贮在此。它们联合在一起，构成自动机的内部状态。(ii) 运控装置: 所有的类和元类(metaclass)全体起着自动机的转移/输出函数的作用。各类以继承关系(只考虑单继承)形成一棵树，树的根是"对象类"；树中含有"函子类*"，它是对象类的子类。相应的各元素以同样的继承关系形成类似的一棵树。(iii) 输入讯道: 接收类和元类的定义以及对象的直接定义; 接收外来的待求解的计算式(含消息传递)。(iv) 输出讯道: 给出对话式的反应信息及求解的结果。

§6. 类作为泛函

我们将类作为在FFP这一核心语言之上的模块设施，它是: ①类似于ADT(抽象数据类型)那样的模块，但是②具有自己的内部状态，从而，比ADT性能还要强些、表达能力更丰富有力的自动机模板; 以及③赋以性能上和实现上的继承机制。既然如此，FFP语言系统应起到自己的作用: 一些基本数据类型的实例(例如，5, "string", symbolicCostant 等)就无需对待成类的实例(即Ω对象)。从而，也就不需要定义自然数类，字符串类，符号常元类等诸如此类

* 函子类将另文讨论

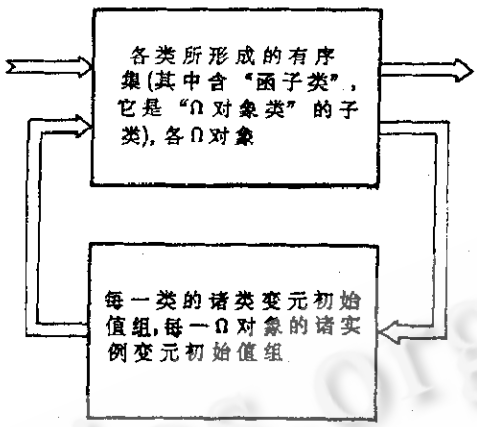

图2　OOPP 的自动机总体结构

的繁琐的类。

OOP 语言的总体结构可理解成一个自动机，该自动机实质上又是由各自动机所构成的一自动机网络，其中各个自动机是元类、类和Ω对象。具体些说，类是自动机模板，它的实例(自动机)是Ω对象。类本身又作为元类的Ω对象，不言而喻，元类是自动机模板，虽然这个模板的实例只有一个，即相关的类。

类作为自动机模板，(对于各实例变元而言)可不配备实际的内部状态。但是，它又是Ω对象，因而，需要配备实际的内部状态(各类变元所取值组)。如果让类的“类变元值组”栖息在FFP-AST 系统的D 中，仅把对于类变元值组的索引记载在类的定义里，那么，类就可以当成泛函来对待。继承通过对泛函所形成的泛函型的调用来体现。

为简单起见，类与其相关的元类* 不予分开，联合在一起成为一个泛函。之所以能够这样做，只要对一个消息传递的接收者，能够识别它是类、还是(它的实例)Ω对象。

类名ClassName 将定义为一个泛函，它取两个函数参量objectName(该参量可能是符号常元cl, 表示类本身)* 和selector(消息传递中的函数)。按照元合成法则[1], 有: <ClassName, objectName, selector>: arguments=ClassName: <<ClassName, objectName, selector>, arguments>, 其中arguments 为消息传递中的诸变目，它采取<arg1, arg2, ...> 或 <ClassName, SubClassName(或 ClassName), arg1, arg2, ...>.

定义几个辅助函数，它们是通用的:

Def inputc≡[1 ∘ 1, 2 ∘ 1, 3 ∘ 1, 2]

Def var≡**apply** ∘ [[%COMP, [%FETCH, 2 ∘ 1], %THREE], 2]/* 它使用在类函数、实例函数的表达中，那时，从诸“变元与其值偶”的序列x 中取某一变元y 的值，

* 在Smalltalk-80 实现中，元类随着类的创建而自动创建[7]。

* 在另一种设计中，不需要此参量。

```
将表达成<var, y>:x, 即(↑ y) ∘ 3:x */
Def beginClass≡id
Def endClass≡(eq∘[2 ∘ 1, %cl]→/* 类本身是消息传递接收者*/
    (not ∘ eq ∘ [RecordSelect ∘ [3 ∘ 1, 2], % ⊥]→
    apply ∘ [RecordSeclect ∘ [3 ∘ 1, 2], (eq ∘ [1 ∘ 1, 1 ∘ 4 ∘ 1]→/* 为子类而工作*/
      [tl ∘ tl ∘ 4 ∘ 1, 2 ∘ 4 ∘ 1, apply ∘ [[%ALFA, [%BU, %apndl, %CELL]],
      apply ∘ [[FETCH, 2 ∘ 4 ∘ 1], defs]], 4, 5, 6, 7]; /* 为本类而工作*/
        [4 ∘ 1, 1 ∘ 1, apply ∘ [[%ALFA, [%BU, %apndl, %CELL]],
        (not ∘ eq ∘ [id, % ⊥]→id; %<>) ∘ apply ∘ [[%FETCH, 1 ∘ 1], defs]], 4, 5, 6, 7])];
      (not ∘ belongsTo ∘ [3 ∘ 1, RecordSelect ∘ [%classMethods, 5]]→/* 不排斥继承*/
      (eq ∘ [1 ∘ 1, 1 ∘ 4 ∘ 1] → apply ∘ [[4, %cl, 3 ∘ 1], apndl ∘ [4, tl ∘ 4 ∘ 1]];
        apply ∘ [[4, %cl, 3 ∘ 1], [[4, 1 ∘ 1, 4 ∘ 1]]); /* 排斥继承*/% ⊥));
    /* 本类的某Ω对象是消息传递的接收者*/
      (not ∘ eq ∘ [RecordSelect ∘ [3 ∘ 1, 3], % ⊥] →
      apply ∘ [RecordSelect ∘ [3 ∘ 1, 3] (eq ∘ [1 ∘ 1, 1 ∘ 4 ∘ 1]→/* 为子类而工作*/
        [tl ∘ tl ∘ 4 ∘ 1, 2 ∘ 4 ∘ 1, apply ∘ [[%ALFA, [%BU, %apndl, %CELL]],
        apply ∘ [[%FETCH, 2 ∘ 1], defs]], 2 ∘ 1, 4, 5, 6, 7]; /* 为本类而工作*/
        [4 ∘ 1, 1 ∘ 1, apply ∘ [[%ALFA, [%BU, %apndl, %CELL]],
        (not ∘ eq ∘ [id, % ⊥]→id; %<>) ∘ apply ∘ [[%FETCH, 2 ∘ 1], defs]], 2 ∘ 1, 4,
        5, 6, 7])];
      (not ∘ belongsTo ∘ [3 ∘ 1, RecordSelect ∘ [%instanceMethods, 5]]→/* 可继承*/
      (eq ∘ [1 ∘ 1, 1 ∘ 4 ∘ 1]→apply ∘ [[4, 2 ∘ 1, 3 ∘ 1], apndl ∘ [4, tl ∘ 4 ∘ 1]];
      apply ∘[[4, 2 ∘ 1, 3 ∘ 1], [4, 2 ∘ 1, 4 ∘ 1]]; /* 排斥继承*/% ⊥))).
        [2, RecordSelect ∘ [%classMethods, 1], RecordSelect ∘ [%instanceMethods, 1],
        RecordSelect ∘ [%super, 1], RecordSelect ∘ [%inheritanceSuppresing, 1],
        RecordSelect ∘ [%classVariables, 1], RecordSelect ∘ [%instanceVariables, 1]]
```

粗略地说，函数endClass是类的解释程序。有意义的是：该解释程序用核心语言FFP编写成，在FFP-AST系统中是可执行的。函数endClass和beginClass一起，将用作“关键词”。于是，

```
Def ClassName≡/* 一个具体类的定义*/
    endClass ∘ [
    %<RECORD, <<super, SuperClassName>,
            <classVariables, <RECORD, <<y1, φ>, <y2, φ>, ...>>>,
            <instanceVariables, <RECORD, <<x1, φ>, <x2, φ>, ...>>>,
            <classMethods, <RECORD, <<classSelector1, classFunction1>,
                <classSelector2, classFunction2>, ...>>>,
            <instanceMethods, <RECORD, <<instanceSelector1, instanceFunction1>,
                <instanceSelector2, instanceFunction 2>, ...>>>,
                <inheritanceSuppressing /* 用来抑制某些可继承的属性、性能*/,
                  <RECORD, <<classVariables, <yj1, yj2, ···>>,
                    <instanceVariables, <xi1, xi2, ...>>,
                    <classMethods, <classSelectorp1, classSelectorp 2, ...>>,
          <instanceMethods, <instanceSeclectorq1, instanceSelectorq2, ...>>
    >> >>,
    inputc
```

] ∘ beginClass

类是元类的实例，需(对类变元)予以初始化：通过<ClassName, cl,initializeClassName> <b1, b2, ...>, 其中initializeClassName 是类 ClassName 的一个类函数选择子；b1, b2, ... 各为类变元 $y_1, y_2, \ldots$ 的初始值。每一类函数均作用于对象 <arguments, ClassName, <<CELL, y1, b1> <CELL, y2, b2>, ...> 或 <>, SuperClassName, <RECORD, <<classVariables, <y_{j_1}, y_{j_2}, ...>>,...,...>>, <RECORD, <<y1, ϕ>, <y2, ϕ>, ... >> , <RECORD, <<x1, ϕ >, <x2, ϕ >, ... >>> 上，对第三成分中 yi 值 bi 的选择，使用 <var, yi>。注意，<b1, b2, ...> 应包括“本类的各类变元及各超类中可继承而未受抑制的各类变元”所应取的各该初始值，而且，b1, b2, ... 的顺序应与“本类各类变元在类(ClassName)定义中所列的顺序，接着，(各向上溯的)超类(可继承而未受抑制的)各类变元在(各该向上溯的)超类定义中所列的顺序”相一致。

Def initializeClassName≡[[2, % “is initialized”],

apply ∘ [[%STORE, 2], [1, defs]]。

(eq ∘ [3, %<>] → [tl ∘ RecordWith ∘ [[%RECORD, 1], trans ∘ [α1 ∘ 1, 2]]∘ [concatenate ∘ [tl ∘ 6, delete ∘ [RecordSelect ∘ [%classVariables, 5], apply ∘ [[%FETCH, 4], defs]]], 1], 2]; [trans ∘]α2 ∘ 3, 1], 2])

为了增加类的隐蔽性，类函数(选择子)可不分离地用定义式来定义，而将定义式(例如，上述最后定义的右边)填写在类(ClassName)定义式的相应的 classFunctioni 的位置，该位置的(函数)选择子即 classSelectori(例如，initializeClassName)。

§7. OOP 对象作为泛函

本文Ω对象这一概念扩展了 Smalltalk-80[7] 的对象，使得同一类的各Ω对象在一定程度上具备自我进化的能力，具体些说，Ω对象不仅具有与其相关的类的全部性能(实例方法)，而且还可形成自己的个性(私有方法)。因此，同一类的各Ω对象可能进化得性能各异。

Ω对象可由三种不同方式创建：由函数定义创建，在线地插入计算式中，用“new”方法动态地创建。按照构造(创建)的方式，Ω对象分为三种：静态Ω对象，插入式Ω对象，动态Ω对象。

7.1. 静态Ω对象。

Ω对象名 objectName 将定义为一个泛函，它取一个函数参量 selector。按元合成法则，有：

<objectName, selector>: arguments

≡ objectName: <<objectName, selector>, arguments>.

Def beginObject≡id

Def inputo≡[1 ∘ 1, 2 ∘ 1, 2]

Def endObject≡(not ∘ eq ∘ [RecordSelect ∘ [2 ∘ 1, 4], % ⊥]→

/* 执行私有函数*/apply ∘ [RecordSelect ∘ [2 ∘ 1, 4],

```
    [2, 1, apply ∘ [[%ALFA, [%BU, %apndl, %CELL]],
    (not ∘ eq ∘ [apply ∘ [[%FETCH, 1 ∘ 1], defs], % ⊥]→
    apply ∘ [[%FETCH, 1 ∘ 1], defs]; %<>)], 3]
    /* 执行类的实例函数*/apply ∘ [[2, 1 ∘ 1, 2 ∘ 1], 3 ∘ 1]) ∘
    [2, RecordSelect ∘ [%Class, 1], RecordSelect ∘ [%instanceVariables, 1],
    RecordSelect ∘ [%privateMethods, 1]]
/* 它可视为Ω对象的解释程序*/
Def objectName≡
    endObject ∘ [
    % <RECORD, <<class, ClassName>,
      <instanceVariables, <x1, x2, ...>>,
      <privateMethods, <RECORD, <<privateSelector1, privateFunction1>,
          <privateSelect2, privateFunction2>, ... >>
    >>> ,
    inputo
    ] ∘ beginObject
```

Ω对象也需予以初始化: 通过<objectName, initializeObject>:<a1,a2,…>,其中initializeObject是与objectName相关联的那个类的一个实例函数选择子; a1, a2, ...各为(相关联的类及诸超类的)实例变元(它们也即该Ω对象的实例变元)x1, x2, ...的初始值。

设计私有函数依据它所作用的对象为<arguments, ClassName, <<CELL, x1, a1>, <CELL, x2, a2>, ...> 或 <>, <x1, x2, ...>>, 对其中第三成分中xi值ai的选择, 使用<var, xi>。

7.2. 插入式Ω对象

在计算式中出现的插入式Ω对象总是与一组变目在一起, 组成一个"作用", 它是"对objectName这一泛函所形成的函数型"的一次调用: <objectName, selector>: argument. 可见, 插入式Ω对象实质上是"接收者为该Ω对象的消息传递"。

7.3. 动态Ω对象。

动态Ω对象靠泛函dynamicObject创建, 再通过泛函dynamicUse动态地使用。按元合成法则, 有:

```
    <dynamicObject, objectName, ClassName>:
      <listOfValuesOfInstanceVariables, listOfPairOfPrivateSelectorsAndFunctions>
      =dynamicObject: <<dynamicObject, objectName, ClassName>, <...,...>>
Def dynamicObject≡apply ∘ [[3 ∘ 1, %cl, %new], [2 ∘ 1, 2]]
/* 即 <ClassName, cl, new>: <objectName, <..., ...>>, 其中new为类ClassName中的类函数选择子*/
Def new≡[[1 ∘ 1, %  "is created"],
    apply ∘ [[%STORE, [%object, 1 ∘ 1]], [[%COMP, %endobject, [%CONSTRUCTN,
    [%RECORD, [[%class, 2] , [%instanceVariables, α1 ∘ tl ∘ 6],
    [%privatemethods, apndl ∘ [%RECORD, 2 ∘ 2 ∘ 1]]]], % inputo],
    %beginObject], apply ∘ [[%STORE, 1 ∘ 1],
    [trans ∘ [α1 ∘ tl ∘ 6, 1 ∘ 2 ∘ 1], defs]]]]
```

Def dynamicUse≡
/*<dynamicUse, objectName, selector>: arguments=dynamicUse: <<dynamicUse, objectName, selector>, arguments>*/
[apply ∘ [apply ∘ [[%FETCH, [%object, 2 ∘ 1]], defs], [[2 ∘ 1, 3 ∘ 1], 2],
apply ∘ [[%pop, 2 ∘ 1], apply ∘ [[%pop, [%object, 2 ∘ 1]], defs]]]

§8. OOP 语言嵌入 FFP-AST 系统[1]

FFP-AST 系统所要求的规定如下:

· 偶 <key, input> 中的 key≡oop.

· **Def** is-oopStream≡eq ∘ [%oop, id]

· 函数 subsystem 的定义中加入一条:

is-oopStream∘ ↑ KEY → ↑ INPUT;

其中 input 取下述五种形式之一:

(i) <ClassName, cl,initializeClassName>: arguments.

(ii) <objectName, initializeObject>: arguments.

(iii) <dynamicObject, objectName, ClassName>: arguments.

(iv) [“不涉及动态Ω对象使用的FFP表达式” ∘ 1, 2] ∘ “使用动态Ω对象的FFP表达式” ∘ “不涉及动态Ω对象使用的FFP表达式”，这里“使用动态Ω对象的FFP表达式”即 <dynamicUse, objectName, selector>: arguments.

(v) “不涉及动态Ω对象使用的FFP表达式”

不言而喻，输入前，所有定义中及input中的FP表达式均需转换成等价的FFP表达式。

§9. 总结

本文以FFP作为核心语言，在FFP-AST系统内相当简单地解决了第一节中提出的三个问题。实际上，我们运用FFP开发了一种兼备FPP和OOPP的语言。有意义的是，这个OOP语言已经是可执行的，它的解释程序就是“定义成FFP表达式的那些关键词”及一些定义泛函。

本文的OOPP具备了Smalltalk-80的OOPP的主要功能，而且还扩充以:

· 继承抑制的功能,

· Ω对象可配备自己的私有函数。

下述一组“公式”是解决本文所提出的问题的方法论基础:

ADT+ 内部状态 = 类 = 自动化(模板),

自动机(模板)+ 继承机制 = 自动机网 = 类网,

(初始化了的)自动机 + 继承机制 =Ω对象。

这一方法论思想能够相当适切地在FFP这样一种十分简朴而又纯泛函的语言中体现出来。这正是本文能简单而颇为完满地解决问题的实质所在。

参考文献

[1] J. Backus, Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs, Comm. of the ACM, 21:8 (1978), 613-641.

[2] 江明德，柳健夫，泛函程序设计语言FFP的$\gamma\omega$演算语义，科学通报，1989年，第21期，第1686-1688页。

[3] 江明德，$\gamma\omega$演算的语义学，计算机学报，1988年，第8期，第449-456页。

[4] 江明德，菊燕，对象作为自动机及面向对象程序设计风格作为自动机网构形，小型微型计算机系统，1988年，第10期，第1-6页。

[5] 舒敏，正式函数程序设计语言FFP在微机上的实现，成都电讯工程学院学报，1987年，第3期，第276-286页。

[6] 李广星，一个函数程序设计系统的研究，硕士学位论文，成都电讯工程学院，1986年2月。

[7] A. Goldberg et al., Smalltalk-80: The Language and its Implementation, Addison-Wesley Publishing Company, 1983.

《系统软件研讨会》征文通知

中国计算机学会软件专业委员会系统软件学组将于1991年召开第三次学术研讨会，现将有关征文内容及注意事项通知如下：

一、征文内容

* 操作系统、编译系统国产化工作的成果与经验
* 计算机语言学，形式语言和语义理论、程序设计逻辑、软件开发形式化技术、软件自动生成技术
* 微机系统软件的开发及汉化工作
* 引进系统的消化、分析、开发与创新
* 大型软件的管理与维护
* “八五”系统软件国产化工作展望

二、征文要求及方法

应征论文要反应本单位或个人实践或理论研究成果，学术上要有严谨的科学性和一定的实用价值。文章论点要明确，文字精练，数据可靠，图表清晰并附有文摘。已在全国性会议，国内外刊物上发表过的文章，请勿应征。征文一律不退稿，请作者自留底稿，截稿时间为1990年12月底，稿件请寄往：

上海市800-209信箱 阮建明

邮政编码 201800

会议时间地点另行通知。

中国计算机学会软件专业委员会系统软件学组

机电部三十二所学术委员会