类型系统 λ演算 面向对象 软件开发 ②

计算机科学 2000Vol. 27№. 5

类型系统的研究与进展*)

5-13

Researches and Advances in Type System

周晓聪 李文军 李师贤 TP311.52

(中山大学计算机科学系 广州 510275)

Abstract In recent years, type system is one of the most important research of theotical computer science. Based on discussing the general concept of type system, this paper outlines the main contents of type system from programming and logical view, and presents the new results in a systematic way.

Keywords λ calculus, Type system, Programming type system, Logical type system

1 引言

类型系统源于罗素为避免朴素集合论中的悖论而引入的"分类"思想。后来邱奇在他的λ演算中也引入了"类型"。60年代初出现的Algol语言提出了数据类型的概念。逻辑学家J. Girard和计算机科学家J. Reynold在70年代初为类型系统引入了"多态性",分别提出了System F[1]和多态λ演算[2]。另一方面,Martin-Lof为了研究数学的逻辑基础,在70年代初提出了直觉类型理论[3],后来也用于程序开发的研究[4]。

80年代后,类型系统的研究更是蓬勃发展,在程序语言设计、程序开发和验证以及机器定理证明方面得到了广泛的应用。目前随着面向对象思想逐渐占据软件开发的主流,面向对象程序设计语言的类型系统逐渐成为人们研究的热点。在80年代初,L. Cardelli提出了"子类型"(subtype)的概念[5],在1985年,他结合多态性,提出了受限量词(bounded quantification)类型)[6],从此这个概念在面向对象类型系统的研究中占据核心地位。90年代以来,人们为了寻求面向对象思想的形式理论基础,一大批学者,象L. Cardelli, B. Pierce, J. Mitchell, M. Abadi, K. Bruce等,对封装、继承、多态、抽象数据类型、子类型等面向对象思想中的核心概念在类型系统中的表示做了大量的研究工作,提出了受限全称类型、受限存在类型、子类型、交类型(intersection type)、并类型(union type)等概念,出现了System F的许多扩充形式的类型系统[7~10],以及各种形式的对象演算系统[11~14]。B. Pierce在他正在写的新书"Type System"[15]中对类型系统研究中的重要事件列成了一张表(有删节):

时 间	事 件
本世纪初	罗素、怀特黑的《数学原理》
30年代	邱奇的无类型λ演算
1940,1958	邱奇的简单类型化λ演算
50年代末	Algol语言
70年代初	Martin-Lof的直觉类型理论
1969	Curry-Howard同构
1972	J. Girard的System F,F$^{\omega}$
1974	J. Reynold的多态λ演算
70年代中	Edinburgh的LCF和ML
1978	交类型
80年代初	子类型
1984	ADT作为存在类型
80年代中	构造演算
80年代中	线性逻辑
1985	受限量词
1987	Edinburgh的逻辑框架
80年代末	纯类型系统
80年代末	扩充的构造演算
1990-1993	高阶子类型
1994-1996	对象演算

*)高等学校博士学科点专项科研基金资助课题,资助号:99-018-411703

本文试图对类型系统研究中的重要成果作简单的介绍和评论，主要包括：类型系统的一般思想、程序类型系统、逻辑类型系统（上表中用楷体表示的概念或系统则或多或少地来源于逻辑）、类型系统应用的研究等。

2 基本思想

广义地说，凡是涉及到"分类"思想的东西都可看成是类型系统研究的内容。抽象地来看，类型系统研究的基本思想包括以下几点：1）使用结构化规则构造所研究的"实体"；2）某些实体作为"类型"，它们中的每一个用来命名其他一些实体的某个特性；3）实体的构造规则和类型化规则可能结合在一起；4）实体可能进行分层，用高一层次的实体作为低一层次的实体的类型。这里所谓的"实体"就是要研究的对象，而"类型"则是一些特殊的实体，用来命名实体的性质。类型系统研究的核心是实体的构造规则和类型化规则及其性质。

用类型系统中的术语来说，所要研究的对象称为"项"，或更准确地说，在类型化以前称为"初始项"(raw term)或"伪项"，而类型化的项称为"已证明项"或"合式项"(well-formed term)。项的构造和类型化可能要在某个环境或上下文中进行，用判断 $\Gamma \vdash M:\tau$ 来表示项 M 在上下文 Γ 下具有类型 τ。项的构造规则和类型化规则以自然演绎的形式给出，通常包括项的引入、消除和相等规则。而对于规则性质的研究，主要针对类型系统的操作语义，包括类型化的可判定性以及定义一些约简，如 β 约简、η 约简等，研究在这种约简下的规范项，规范项被看成一般项的"值"，并研究项的强规范性，其含义是项在有限次约简后能得到规范项。

根据研究的侧重点不同，类型系统大致可分为程序类型系统和逻辑类型系统。程序类型系统主要借鉴 λ 演算的思想，侧重于研究类型系统的计算性质，目的是要表示程序中数据类型和可计算的函数，主要应用在程序语言的设计方面，其代表是多态 λ 演算[2]、带子类型系统[10-16]和对象演算系统[11~13]。逻辑类型系统则以直觉主义逻辑为基础，侧重于研究类型系统的逻辑性质，目的是要表示各种各样的逻辑演算系统，主要应用在定理自动证明和程序自动开发与验证方面，其代表是直觉类型理论[3]、构造演算[18]和扩充的构造演算[17]。当然这种划分不是绝对的，只是为了帮助我们对整个类型系统的研究有更为清晰的了解。实际上，类型作为程序的规范和作为逻辑公式有着深刻的联系，同样项作为满足规范的程序和作为逻辑公式的证明也是有着密切的关系。

3 程序类型系统

简单类型化的 λ 演算是程序类型系统研究的起点，提供了一个基本的研究框架，如类型和项的构造、上下文、类型化规则、规则的性质、约简规则及其性质等，是每个程序类型系统都必须研究的。多态 λ 演算的提出是程序类型系统研究的重要里程碑，随着面向对象思想的广泛应用，子类型、受限量词和对象的表示等在类型系统的研究中占据越来越重要的地位，近年提出的对象演算系统为研究面向对象的类型系统提供了新的研究思路和方向。

3.1 简单类型化 λ 演算

所有的类型系统都要用到 λ 演算中的基本思想。λ 演算研究的对象被称为项，可用类 BNF 描述如下：

$$M,N ::= x \mid \lambda x M \mid MN$$

其中 x 取至一个集合 V，被称为项变量，或简称变量，所以 λ 演算是建立在一个项变量集合上的演算系统。λ 演算是函数式程序设计语言的理论基础，它本身也可看成一个语言，而 β 约简就是它的操作语义，β 规范项被看成所求的值。

在 λ 演算中引入类型有两种方法，一种是 Curry 式，其特点是项中不附带类型信息，被认为是类型指派系统，类型化规则可看成给项指派类型的规则。另一种被称为 Church 式，其特点是项中附带类型信息，类型化规则不仅给项指派类型，而且还给出项的构造方式。简单类型化 λ 演算通常记为 $\lambda\!\rightarrow$ 演算，其引入类型的思想是选定一个基类型集合 T，该集合可只有一个元素。$\lambda\!\rightarrow$ 演算中研究的对象包括类型和项，用类 BNF 可描述为：

$\alpha,\beta ::= \tau \mid \alpha \rightarrow \alpha, \tau \in T$ 是基类型

$M,N ::= x \mid \lambda x : \alpha . M \mid MN$

上面是 Church 式的类型化 λ 演算，而 Curry 式 λ 演算的区别在于它的项与无类型的 λ 演算完全相同，只是引入类型化规则。为说明 $\lambda\!\rightarrow$ 演算的类型化规则，需先定义上下文 Γ，它是一个变量：类型的表，构造方式为：

$\Gamma ::= [\,] \mid \Gamma, x : \alpha$

其中[]代表空表，而 x 是变量，α 是类型

$\lambda\!\rightarrow$ 演算中的类型规则是形如 $\Gamma \vdash M : \alpha$ 的判断，表示在假设 Γ 下，项 M 的类型为 α，这里上下文 Γ 的含义就是要给项 M 中的每个自由变量指派一个类型，类型系统中的类型化规则大都是以一种自然推理的形式给出，具体说，就是根据类型的构造，给出每种类型下项的引入、消除等规则，如果要研究项是否相等，则还引入等式规则，包含等式规则的类型系统被称为类型理论。在 $\lambda\!\rightarrow$ 演算中，类型的构造只有 $\rightarrow$ 一种方式，因此 $\lambda\!\rightarrow$ 演算中类型化的规则最重要的是 $\rightarrow$I 规则和 $\rightarrow$E 规则，形式为：

$$\rightarrow\text{I}\quad \frac{\Gamma, x:\alpha \vdash M:\beta}{\Gamma \vdash \lambda x:\alpha.M:\alpha\rightarrow\beta}$$

$$\rightarrow\text{E}\quad \frac{\Gamma \vdash M:\alpha\rightarrow\beta \quad \Gamma \vdash N:\alpha}{\Gamma \vdash MN:\beta}$$

对于 λ→ 演算中类型化规则首先是研究结构性质，这些性质与项和类型的构造结构有关，主要的结构性质包括：设有 $\Gamma \vdash M:\alpha$，则

1. M 中所有自由变量都出现在 Γ 中，

2. 排列规则，其含义是，变换 Γ 中变量的顺序，上述判断仍成立。

3. 弱化规则，其含义是，在 Γ 中增加 M 没有的自由变量，判断仍成立。

4. 强化规则，其含义是，去掉 Γ 中不出现在 M 中的自由变量，判断仍成立，

5. 替换规则，其含义是，若又有 $\Gamma, x:\alpha \vdash N:\beta$，则有 $\Gamma \vdash N[M/x]:\beta$。

6. 类型唯一性，其含义是，若又有 $\Gamma \vdash M:\beta$，则有 $\alpha=\beta$。

对于类型化规则，需要研究一些判定问题，即是否有可终止的算法判定下面的问题：

1. 类型检查问题，给定 M 和 α，是否有 Γ 使得：$\Gamma \vdash M:\alpha$?

2. 类型推导问题，给定 M，是否有 Γ 和 α 使得：$\Gamma \vdash M:\alpha$?

3. 类型下有否项(inhabitation)的问题，给定一个类型 α，是否有 M，使得 $\vdash M:\alpha$?

在 λ→ 演算中仍定义 β 约简作为它的操作语义，这时除研究 Church-Rosser 性质、β 规范式、强规范性以外，还包括研究 β 约简是否保持项的类型，这个性质通常被称为主题约简，即若 $\Gamma \vdash M:\alpha$，且 $M\rightarrow_\beta N$，是否有 $\Gamma \vdash N:\alpha$，

3.2 多态 λ 演算

类型系统中的多态性概念在 70 年代初被引入，有了多态性的 λ 演算称为多态 λ 演算，或称为 System F，在 λ-方阵(λ-cube)也被记为 λ2 演算。类型系统中的多态性的基本含义是一个项可能具有多个类型，其特点首先是引入类型变量，其次是引入量词类型，主要是全称量词。这时类型和项的构造方式用类 BNF 可描述为：

1. $\alpha,\beta ::= X \mid \alpha\rightarrow\alpha \mid \forall X.\alpha$，其中 X 是类型变量；

2. $M,N ::= x \mid \lambda x:\alpha.M \mid MN \mid \Lambda X.M \mid M\alpha$。

λ2 演算建立在两个变量集合上，分别称为类型变量集和项变量集，通常假设这两个集合不相交。λ2 演算中上下文 Γ 不仅包括项变量及其类型，还包括类型变量，其构造方式可用类 BNF 描述为：

$$\Gamma ::= [] \mid \Gamma, X \mid \Gamma, x:\alpha$$

实际上作为一个合式的上下文还需要其他一些条件。

这时类型的构造包括了多态类型 $\forall X.\alpha$，因此类型化规则增加下面两个规则：

$$\forall\text{-I}\quad \frac{\Gamma, X \vdash M:\alpha}{\Gamma \vdash \Lambda X.M:\forall X.\alpha}$$

$$\forall\text{-E}\quad \frac{\Gamma \vdash M:\forall X.\alpha}{\Gamma \vdash M\beta:\alpha[\beta/X]}$$

对于 λ2 演算的扩充是引入类别(kind)的概念，使得量词不仅可约束在代表类型个体的变量上面，还可约束在代表类型函数(指从类型到类型的函数，称为算子)的变量上，从而得到高阶多态 λ 演算，记为 λω 演算，或称为 System Fω，对应地 λ2 演算称为二阶多态 λ 演算。

λω 演算的特点是引入类别，作为类型的类型，算子的类型等，其中有一个特定的类别是所有类型的类别，下面记为 Type。这样 λω 演算中研究的对象就包括类别、算子(类别为 Type 的算子仍可称为类型)和项，它们的构造方式用类 BNF 可描述为：

1. 类别：$K,L ::= \text{Type} \mid K\rightarrow L$；

2. 算子：$\alpha,\beta ::= X \mid \Lambda X:K.\alpha \mid \alpha\beta \mid \alpha\rightarrow\alpha \mid \forall X:K.\alpha$；

3. 项：$M,N ::= x \mid \lambda x:\alpha.M \mid MN \mid \Lambda X.M \mid M\alpha$。

这时上下文的构造方式用类 BNF 可描述为：

$$\Gamma ::= [] \mid \Gamma, X:K \mid \Gamma, x:\alpha$$

在 λω 演算中，首先有算子的类别化规则，该规则与 λ→ 演算中的类型化规则类似，但要保持 $\alpha\rightarrow\alpha$ 和 $\forall X:K\alpha$ 的类别是 Type。λω 演算中最重要的类别化规则包括：

$$\text{K-Fun-I}\quad \frac{\Gamma, X:K \vdash \alpha:L}{\Gamma \vdash \Lambda X:K.\alpha:K\rightarrow L}$$

$$\text{K-Fun-E}\quad \frac{\Gamma \vdash \alpha:K\rightarrow L \quad \Gamma \vdash \beta:K}{\Gamma \vdash \alpha\beta:L}$$

$$\text{K-Arrow}\quad \frac{\Gamma \vdash \alpha:\text{Type} \quad \Gamma \vdash \beta:\text{Type}}{\Gamma \vdash \alpha\rightarrow\beta:\text{Type}}$$

$$\text{K-Poly}\quad \frac{\Gamma, X:K \vdash \alpha:\text{Type}}{\Gamma \vdash \forall X:K.\alpha:\text{Type}}$$

有了类别化规则之后，上下文 Γ 的构造更严格地说，只有当 $\Gamma \vdash \alpha:\text{Type}$ 时，$\Gamma, x:\alpha$ 才是合式的上下文。λω 演算中的类型化规则与 λ2 演算类似，其特点是只有类别 Type 的算子下才有项。这时的 ∀-I 和 ∀-E 规则分别为：

$$\forall\text{-I}\quad \frac{\Gamma, X:K \vdash M:\alpha}{\Gamma \vdash \Lambda X:K.M:\forall X:K.\alpha}$$

$$\forall\text{-E}\quad \frac{\Gamma \vdash M:\forall X:K.\alpha \quad \Gamma \vdash \beta:K}{\Gamma \vdash M\beta:\alpha[\beta/X]}$$

3.3 子类型和受限量词

子类型的思想在程序设计语言中很早就被使用，例如大多数语言都很自然地认为整数可被看成实数，能参加所有实数能进行的操作。对子类型的概念在理论上作严格的考察则开始于 L. Cardelli 在 80 年代初

对继承的语义的研究[5]，L. Cardelli 等人在 80 年代中又提出了受限量词的概念[6]，将子类型与多态结合在一起，使得子类型和多态成为研究面向对象程序设计语言的类型系统中的核心概念。

子类型是类型系统中类型集上的一个序关系，说类型 $\alpha\leqslant\beta$，意味着凡是要求类型为 β 的项的地方都可使用类型为 α 的项来代替。在类型系统中，子类型有两种不同的途径发挥它的作用。一种是显式转换方式，即如果 $\alpha\leqslant\beta$，那么在类型系统中引入一个特殊的项 $C_{\alpha,\beta}(M)$，其中 M 的类型为 α，而 $C_{\alpha,\beta}(M)$ 的类型则为 β。另外一种是隐式转换方式，即如果 $\alpha\leqslant\beta$，那么若 $\Gamma\vdash M:\alpha$，则有 $\Gamma\vdash M:\beta$，这个规则被称为 Subsumption，可译作项的归类规则，在下面采用这种形式。

引入子类型关系后，通常在相应的演算名称后加脚注 $\leqslant$，例如对于 λ2 演算，相应地记为 $\lambda2_{\leqslant}$。对应的 System F 系统也采用类似的记号。显然在简单类型化的 λ→ 演算中就可引入子类型，这里介绍最复杂的情况，即 $\lambda\omega_{\leqslant}$ 演算，其中的子类型称为高阶子类型。其他的演算可通过对该演算简化而得到，分别称为简单子类型和二阶子类型等。

这里介绍的 $\lambda\omega_{\leqslant}$ 演算包括受限全称量词类型，即在引入全称量词时包含子类型关系，其一般形式为 $\forall X\leqslant\alpha:K.\beta$。具体来说，在 $\lambda\omega_{\leqslant}$ 演算中研究的对象包括了类别、算子、子类型关系和项等，注意这里子类型关系不仅定义在类型上，也定义在一般的算子上。这里类别、算子和项的构造用 BNF 可描述为：

1. 类别：$K,L::=\mathrm{Type}\mid K\to K$；

2. 算子：$\alpha,\beta::=X\mid\alpha\to\beta\mid\Lambda X:K.\alpha\mid\alpha\beta\mid\forall X\leqslant\alpha:K.\beta\mid\mathrm{Top}$；

3. 项：$M,N::=x\mid MN\mid\lambda x:\alpha.M\mid M\alpha\mid\Lambda X\leqslant\alpha:K.M$。

$\lambda\omega_{\leqslant}$ 演算中的上下文包含子类型关系，即在上下文中引入类型变量时的一般形式为 $X\leqslant\alpha:K$，称 α 为类型变量 X 的界声明。上下文的构造方式可描述为：

$\Gamma::=[]\mid\Gamma,X\leqslant\alpha:K\mid\Gamma,x:\alpha$

更严格地，在引入 $X\leqslant\alpha:K$ 和 $x:\alpha$ 时，必需先分别满足条件 $\Gamma\vdash\alpha:K$ 和 $\Gamma\vdash\alpha:\mathrm{Type}$。

注意上述构造规则建立在类别化规则的基础上，$\lambda\omega_{\leqslant}$ 演算的类别化规则与 λω 很类似，只是其中的 K-Poly 规则要改为：

$$\text{K-Poly}\quad\frac{\Gamma\vdash\alpha:K\quad\Gamma,X\leqslant\alpha:K\vdash\beta:\mathrm{Type}}{\Gamma\vdash\forall X\leqslant\alpha:K.\beta:\mathrm{Type}}$$

$\lambda\omega_{\leqslant}$ 演算中的子类型关系是形式为 $\Gamma\vdash\alpha\leqslant\beta:K$ 的判断，其中 α 和 β 都是算子，产生子类型关系的重要规则包括：

$$\text{S-Poly}\quad\frac{\Gamma,X\leqslant\alpha:K\vdash\beta\leqslant\tau:\mathrm{Type}}{\Gamma\vdash\forall X\leqslant\alpha:K.\beta\leqslant\forall X\leqslant\alpha:K.\tau:\mathrm{Type}}$$

$$\text{S-Arrow}\quad\frac{\Gamma\vdash\alpha'\leqslant\alpha:\mathrm{Type}\ \Gamma\vdash\beta\leqslant\beta':\mathrm{Type}}{\Gamma\vdash\alpha\to\beta\leqslant\alpha'\to\beta':\mathrm{Type}}$$

$$\text{S-Fun-I}\quad\frac{\Gamma,X:K\vdash\alpha\leqslant\beta:L}{\Gamma\vdash\Lambda X:K.\alpha\leqslant\Lambda X:K.\beta:K\to L}$$

$$\text{S-Fun-E}\quad\frac{\Gamma\vdash\alpha\leqslant\beta:K\to L\quad\Gamma\vdash\tau:K}{\Gamma\vdash\alpha\tau\leqslant\beta\tau:L}$$

引入子类型之后，对项的类型化规则的修改包括：

$$\text{T-Subsumption}\quad\frac{\Gamma\vdash M:\alpha\qquad\alpha\leqslant\beta}{\Gamma\vdash M:\beta}$$

$$\forall\text{-I}\quad\frac{\Gamma,X\leqslant\beta:K\vdash M:\alpha}{\Gamma\vdash\Lambda X\leqslant\beta:K.M:\forall X\leqslant\beta:K.\alpha}$$

$$\forall\text{-E}\quad\frac{\Gamma\vdash M:\forall X\leqslant\beta:K.\alpha\ \Gamma\vdash\tau\leqslant\beta:K}{\Gamma\vdash M\tau:\alpha[\tau/X]}$$

研究 $\lambda\omega_{\leqslant}$ 演算的性质包括类别化、子类型关系和类型化规则的性质，不仅要研究类别的可判定性问题，也要研究子类型关系的可判定性问题，只有在这两者的基础上才能讨论项的类型化的可判定性。对于带子类型，特别是包括受限量词的类型系统的研究还处于发展之中，许多性质还没得到确定的证明，特别是子类型关系，对其中的规则稍加改动将对整个类型系统的性质产生很大的影响，读者可参考文[16,19,20]等。

3.4 对象的表示与对象演算系统

随着面向对象思想的广泛应用，研究其形式化理论基础显得越来越重要，而类型系统可为面向对象中的许多核心概念，例如类、对象、消息传递、封装、多态、继承、动态联编等提供形式化的描述方法，逐渐成为研究面向对象思想的形式化理论基础的一个重要方面。从 90 年代初以来，面向对象的类型系统一直是人们研究的热点课题。

所有在类型系统中描述对象的方法都基于用记录类型表示对象这样一个基本思想，记录类型可看成是一种带标号的积类型。在用记录类型表示对象和类时，有两种不同方法，一种是只表示对象和类的界面，而不表示对象和类的实现；另外一种是连带表示对象和类的实现，即其私有的数据表示等。通常对前一种方法的研究更多。

在类型系统中研究对象和类的表示从总的方面来说可分为两个途径：一种是在传统的类型系统中表示对象和类，另外一种是将对象作为原子概念加以研究。其中第一种研究途径的代表性工作包括 W. Cook 等人以递归类型为基础的记录演算[21]，B. Pierce 等人的以存在量词类型为基础的记录演算[22]和 G. Castagna 的基于重载和延迟联编的模型[8]等。而第二种途径中最具代表性的工作包括 L. Cardelli 的将对象作为原子概念的对象演算系统[11]和 J. Mitchell 的基于方法特化的对象演算系统[13]。文[14]和[23]对在传统的类型系统(主要是指在二阶或高阶多态 λ 演算)中表示对象的方法做了很好的比较。下面以基于递归类型的记录

类型表示对象方法为例，说明研究面向对象类型系统的基本思想。

用基于递归类型的记录类型表示对象的基本点包括，首先要区分对象类型和类，对象的类型是类型系统中的类型，通常是递归类型，因为对象的方法中，通常含有相同类型的其他对象作为参数，对象作为对象类型的值是类型系统中的项，而类也是类型系统中的项，被看成是创建对象的项。例如表示对象 point 的方法是：

```
Point={ x : int,
        move : int→T,
        equal : T→bool
      }
Object_Point=μT. Point ={μT {x:int,
                             move:int→T,
                             equal:T→bool
                            }
point_def=ΛT≤Point:Type. λself:T. {x=3,
      move=λdx:int,{…self↑x+dx…},
      equal=λother:T. (equalint other↑x self↑x)
      }:∀ T≤Point. T→Point
point=Fix(point_def Object_Point):Object_Point
```

其中 Point 是一个记录类型，Object_Point 是一个递归类型，它是对象 point 的类型，记录类型的值 point_def 用来定义对象 point，对象 point 是项 point_def 的不动点，其中 Fix 是在记录类型用来求项的不动点的组合子。项 equal_int 用来判断两个整数是否相等，其类型是 int→(int→bool)。而相应的类的定义将是：

```
Point_Class=ΛT≤Point!Type
            λmyclass:int→(T→T)
            λinit_x:int. λself:T. {
x=init_x,
move=λdx:int. Fix(myclass(self↑x+dx)),
equal=λother:T. (equal_int other↑x self↑x)
}:∀ T≤Point:Type int→(T→Point)
point_2=Fix(Fix
    (Point_Class Object_Point)(2)):Object_Point
```

直观地看，Point_Class 是用整数来创建具有类型 Object_Point 的对象 point_2的项，因此这里的对象和类是不同的。

其次要区分子类型和继承，子类型是类型之间的关系，而继承是通过一个类和对象，经过扩充和重载之后得到另一个类和对象的机制。继承通过记录类型的连接和域的修改等操作来实现。在用递归类型表示对象和类时，一个对象 A 通过继承（更准确说应该是委派（delegation），对于类才说是继承）另一个对象 B 而得到时，并不能推出对象 A 的类型是对象 B 的类型的子类型，这有两个原因，一是子类型规则对于函数类型来说是反变的，另一个原因是在表示类型时需要用 self 来访问对象自己的方法。

对象在类型系统中的表示以及对象演算系统都还在不断发展之中，许多问题还需要进一步研究。

4 逻辑类型系统

逻辑类型系统的特点是将类型系统看成一个逻辑系统，或相对应地，用类型系统的语言来表达逻辑。著名的 Curry-Howard 对应（correspondence），即“命题看成类型”是将类型系统看成逻辑系统的核心。在逻辑类型系统中 Martin-Lof 的直觉类型理论（ITT）有着非常重要的地位，对于国内研究类型系统的影响也最大。T. Coquand 和 G. Huet 所提出的构造演算（Calculus of Construction，简称 CC 或 λC）是直觉类型理论的扩充，引入了 System F 中的非谓词性，而 Luo Zhaohui 的扩充构造演算（Extended Calculus of Construction，简称 ECC）则将直觉类型理论中的类型系（type universes）和非谓词性结合起来，是 CC 的扩充。纯类型系统[24]试图对形形色色的类型系统作为一个整体统一进行研究，更深入地探讨类型系统中的基本概念，可以看成是第二节所讨论的类型系统基本思想的形式化描述。

4.1 Curry-Howard 对应与直觉类型理论

将类型系统看成是逻辑系统的核心思想是 Curry-Howard 对应，即“命题被看成类型”，进而类型下的项就是该命题的证明。Martin-Lof 在文[3]中详细讨论这种对应关系，并给出其他一些对应关系。设 α 是类型系统中的类型，M 是类型 α 下面的项，则可解释成：

1. α 是一个集合，M 是集合 α 中的一个元素，如果 α 有项存在，说该集合是非空的。

2. α 是一个命题，M 是该命题的一个证明，如果 α 有项存在，说该命题是真的。这就是 Curry-Howard 对应最初的思想。

3. α 是一个目标（expectation），M 是达到该目标的一个可实现的方法，如果 α 有项存在，则说该目标是可实现的。

4. α 是一个问题（程序的规范），M 是解决该问题的方法（程序），如果 α 有项存在，则说该问题（规范）是可解的（可满足的）。

可以看出这些解释不仅是研究逻辑类型系统的基础，也是将逻辑类型系统用于程序的自动开发和验证，以及定理证明的开发的基础。Martin-Lof 的直觉类型理论 ITT 主要采用第一种解释，即类型 α 是一个集合，项 M 是该集合的一个元素。ITT 主要讨论以类型系统的方式来研究集合的构造，所以说 ITT 最初的目标是从直觉主义的基本观点出发，研究一个与公理集合论相容的构造集合理论。

下面就 ITT 提出的几种重要的集合构造方式，作为上面对应关系的具体的例子加以讨论。首先是空集，记为⊥，其逻辑解释是 false，从类型系统角度看，它是没有项的类型。其次是两个集合的笛卡尔积，记为 α&β，逻辑解释是两个命题的“与”，对应类型系统中的积类型。元素是 (M,N)，逻辑解释是 M 是命题 α 的一

个证明，同时 N 是命题 β 的一个证明。在类型系统中积类型下的项是对(pair)。再其次有两个集合的不相交并，记为 $\alpha \vee \beta$，逻辑解释是两个命题的"或"，对应类型系统中的和(或者说共积)类型。元素是 $\langle M,0\rangle$，其中 M 是命题 α 的证明，或者是 $\langle N,1\rangle$，其中 N 是命题 β 的证明。还有蕴含集合，记为 $\alpha \supset \beta$，逻辑解释是命题 α 推出 β，即 $\alpha \Rightarrow \beta$，集合中的元素是 $\lambda x{:}\alpha.M$，即是一个函数，将 α 命题的每个证明 N，产生(构造)命题 β 的一个证明 $M[N/x]$。

上面是一些简单的集合构造方法，而 ITT 中引入的最重要的集合构造是，Π-构造，称为集合簇的笛卡尔积，和 Σ-构造，称为集合簇的不相交并，分别用来对应逻辑系统中的全称量词(∀)和存在量词(∃)，在类型系统中则分别称为依赖积类型和依赖和(共积)类型，这里依赖的含义是指，这两个类型的构造依赖于项。在前面所讨论的(程序)类型系统中，只有类型的构造依赖于类别，项的构造依赖于类型，甚至于类别，但不会反过来，类型的构造依赖于项。具体来说，Π-构造的形式是：

$$\frac{\Gamma \vdash \alpha{:}\mathrm{Set} \quad \Gamma, x{:}\alpha \vdash \beta{:}\mathrm{Set}}{\Gamma \vdash \Pi x{:}\alpha.\beta{:}\mathrm{Set}}$$

这时当上下文 Γ 的一般形式为 $[x_1{:}\alpha_1, x_2{:}\alpha_2, \cdots, x_n{:}\alpha_n]$ 时，类型 α_i 中可能出现项变量 $x_1, \cdots, x_{i-1}$，因此上下文中变量出现的顺序不能随意改变。对应地，元素(项)的构造形式为：

$$\frac{\Gamma, x{:}\alpha \vdash M{:}\beta}{\Gamma \vdash \lambda x{:}\alpha.M{:}\Pi x{:}\alpha.\beta}$$

可以看出，这与前面的 $\rightarrow$I 规则形式类似，实际上当 β 中不出现项变量 x 时，依赖积类型就是函数类型，即当 x 不出现在 β 中，可定义 $\alpha \rightarrow \beta = \Pi x{:}\alpha.\beta$。根据这一点也可给出依赖积类型下项的逻辑解释，集合 $\Pi x{:}\alpha.\beta$ 中的元素都是函数，这个函数作用在命题 α 的任意证明(元素) N，都产生命题 $\beta[N/x]$ 的一个证明 $M[N/x]$。

对应地，集合簇的不相交并的构造形式为：

$$\frac{\Gamma \vdash \alpha{:}\mathrm{Set} \quad \Gamma, x{:}\alpha \vdash \beta{:}\mathrm{Set}}{\Gamma \vdash \sum x{:}\alpha.\beta{:}\mathrm{Set}}$$

而其中元素(项)的构造形式是：

$$\frac{\Gamma \vdash M{:}\alpha \quad \Gamma \vdash N{:}\beta[M/x]}{\Gamma \vdash \langle M,N\rangle{:}\sum x{:}\alpha.\beta}$$

根据元素的构造规则，可以给出集合 $\Sigma x{:}\alpha.\beta$ 的一个逻辑解释，其中的元素都是对 $\langle M,N\rangle$，且 M 是命题 α 的一个证明，对应地 N 是命题 $\beta[M/x]$ 的一个证明。可见当 β 不依赖于变量 x 时，集合簇的不相交并是两个集合的笛卡尔积，即当 x 不在 β 中出现时，可定义 $\alpha \& \beta = \Sigma x{:}\alpha.\beta$。

一个很自然的问题是 Set 本身是否可看成一个集合，在 Martin-Löf 早期的类型理论中，存在一个类型，它是所有类型的类型，即对应有一个集合是所有集合的集合。对此 J. Girard 发现它是不一致的，即每个类型都有项，从逻辑上看，就是每个命题都有证明。这就是有名的 Girard 悖论。在文[3]中，Martin-Löf 引入了类型系，即在上述集合构造方式下封闭的所有集合构成一个集合，由上述方式构造的集合称为小集合，而所有小集合构成的集合不再是小集合，例如上面的 Π-构造中，将其中的 α 改为 Set，即 $\Pi x{:}\mathrm{Set}.\beta$ 不是小集合，不属于 Set，这是所谓的谓词性(predicative)，即将量词作用在变量上之后，得到的不再与该变量所处的层次属于同一层次。如果认为有 $\Pi x{:}\mathrm{Set}.\beta{:}\mathrm{Set}$，则产生 Girard 悖论。

注意前面所给出的多态 λ 演算是非谓词性的，其中有一个类别，即 Type 是所有类型的类型，量词∀ 作用在变量 X:Type 后产生的还是类型，那为什么不产生悖论呢?这是因为在多态 λ 演算中，并没将 Type 与集合完全等同起来，或者更准确地说，是将 Type 解释成一种特殊的泛集合(universal set)，它对于量词∀ 等类型构造手段封闭。

4.2 构造演算与扩充的构造演算

构造性演算(CC)是 ITT 中的依赖积、依赖和等概念与多态 λ 演算中的多态等概念的结合。对于 ITT 来说，它引入了非谓词性，但同样地没有把集合与类型完全等价起来，对于 $\lambda\omega$ 演算来说，其扩充就是引进了依赖积和依赖和，使得类型的构造依赖于项的构造。

可以说，从类型系统中研究的实体来看，逻辑类型系统的特点是不能再完全区分类型和项，因为在这里项的构造依赖于类型，类型的构造也依赖于项，从而所有的实体，甚至包括类别来说，都是项。除了引入依赖积和依赖和之外，CC 与 $\lambda\omega$ 演算的一个区别是它使用的常量类别称为 Prop，其中的项被看成命题(注意这里的命题不仅是命题演算中的命题，还包括谓词演算中带有量词的公式，其实更准确地说是公式)，Prop 具有非谓词性，即：若 $\Gamma, X{:}\mathrm{Prop} \vdash \alpha{:}\mathrm{Prop}$，则 $\Gamma \vdash \Pi X{:}\mathrm{Prop}.\alpha{:}\mathrm{Prop}$，但 Prop 本身并不是 Prop，则不能导出判断：$\vdash \mathrm{Prop}{:}\mathrm{Prop}$。

扩充的构造演算(ECC)是 CC 的扩充，体现在它引入了类型系，使得不仅有类别常量 Prop，还有 $\mathrm{Type}_0, \mathrm{Type}_1, \cdots, \mathrm{Type}_n, \cdots$ 等无穷多个类别常量，且有：

$$\mathrm{Prop} \leq \mathrm{Type}_0 \leq \mathrm{Type}_1 \leq \cdots \leq \mathrm{Type}_n \leq \cdots$$

其中 $\leq$ 是在项上定义的一个关系。只有 Prop 是非谓词的，而其他的类别常量都是谓词的。

ECC 中项由下面的子句递归地定义：

1. 类别常量 $\mathrm{Prop}, \mathrm{Type}_0, \mathrm{Type}_1, \cdots, \mathrm{Type}_n, \cdots$ 等

是项；

2. 变量 $x,y,\cdots$ 是项；

3. 如果 M,N 和 A 是项，则 $\Pi x:M.N,\lambda x:M.N$，$MN,\Sigma x:M.N,\mathrm{pair}_A(M,N),\pi_1(M),\pi_2(M)$ 都是项。

上下文的一般形式是 $\Gamma=[x_1:A_1,\cdots,x_n:A_n]$，其中 x_i 是变量，而 A_i 是项。判断的形式为 $\Gamma\vdash M:A$，其中 M 和 A 都是项。判断由下面的规则产生，其中的 j 代表任意的自然数：

Ax $\dfrac{}{\vdash \mathrm{Prop}:\mathrm{Type}_j}$

C $\dfrac{\Gamma\vdash A:\mathrm{Type}_j}{\Gamma,x:A\vdash \mathrm{Prop}:\mathrm{Type}_0}$

T $\dfrac{\Gamma\vdash \mathrm{Prop}:\mathrm{Type}_0}{\Gamma\vdash \mathrm{Type}_i:\mathrm{Type}_{i+1}}$

var $\dfrac{\Gamma,x:A,\Gamma'\vdash \mathrm{Prop}:\mathrm{Type}_0}{\Gamma,x:A,\Gamma'\vdash x:A}$

$\Pi 1$ $\dfrac{\Gamma,x:A\vdash P:\mathrm{Prop}}{\Gamma\vdash \Pi x:A.P:\mathrm{Prop}}$

$\Pi 2$ $\dfrac{\Gamma\vdash A:\mathrm{Type}_j\quad \Gamma,x:A\vdash B:\mathrm{Type}_j}{\Gamma\vdash \Pi x:A.B:\mathrm{Type}_j}$

λ $\dfrac{\Gamma,x:A\vdash M:B}{\Gamma\vdash \lambda x:A.M:\Pi x:A.B}$

app $\dfrac{\Gamma\vdash M:\Pi x:A.B\quad \Gamma\vdash N:A}{\Gamma\vdash MN:B[N/x]}$

$\sum$ $\dfrac{\Gamma\vdash A:\mathrm{Type}_j\quad \Gamma,x:A\vdash B:\mathrm{Type}_j}{\Gamma\vdash \Pi x:A.B:\mathrm{Type}_j}$

pair $\dfrac{\Gamma\vdash M:A\quad \Gamma\vdash N:B[M/x]\quad \Gamma,x:A\vdash B:\mathrm{Type}_j}{\Gamma\vdash \mathrm{pair}_{\Sigma x:A.B}(M,N):\Sigma x:A.B}$

$\pi 1$ $\dfrac{\Gamma\vdash M:\Sigma x:A.B}{\Gamma\vdash \pi_1(M):A}$

$\pi 2$ $\dfrac{\Gamma\vdash M:\Sigma x:A.B}{\Gamma\vdash \pi_2(M):B[\pi_1(M)/x]}$

$\leq$ $\dfrac{\Gamma\vdash M:A\quad \Gamma\vdash A':\mathrm{Type}_j\quad A\leq A'}{\Gamma\vdash M:A'}$

从规则 $\Pi 1$ 可看出 Prop 的非谓词性，即可有：$\vdash \Pi x:\mathrm{Prop}.x:\mathrm{Prop}$，实际上在 ECC 中该项代表 false。从 $\Pi 2$ 和 Σ 可看出 Type_i 的谓词性，即若有 $\Gamma\vdash \Pi x:A.B:\mathrm{Type}_i$，则 A 不可能是 Type_i 本身。

文[17]中证明了 ECC 的一系列性质，包括 ECC 中的 Γ 项是强规范的，即如果有判断 $\Gamma\vdash M:A$，则 M 是强规范的。而且 ECC 中的类型检查是可判定的，作为逻辑系统，ECC 是一致的，即不存在 M 使得 $\vdash M:\Pi x:\mathrm{Prop}.x$。

4.3 纯类型系统

90年代初纯类型系统被 H. Barendregt 提出来加以研究，开始时称为通用类型系统。纯类型系统的基本思想是以项之间的依赖性为核心，更抽象地研究类型系统及其性质。在纯类型系统的研究中，首先要给出类型系统所研究的对象（实体）是什么，因此定义下面的初始项集：

$$A::=V\mid C\mid AA\mid \lambda V:A.A\mid \Pi V:A.A$$

其中 V 是变量集，C 是一个常量集，这里只考虑 λ 抽象和依赖积类型，当然也可引入依赖和类型等，读者可参考文[25,26]。

定义1 一个 PTS 的规范（Specification）是一个三元组 $\mathrm{Spec}=(S,A,R)$，其中：

1. S 是 C 的一个子集，被称为类子（sorts）。

2. A 是一些公理的集合，其中每个公理具有形式：$c:s$ 其中 $c\in C,s\in S$。

3. R 是一些规则的集合，其中每个规则具有形式：(s_1,s_2,s_3)，其中 $s_1,s_2,s_3\in S$，当 $s_2=s_3$ 时，简写为 (s_1,s_2)。

定义2 被规范 $\mathrm{Spec}=(S,A,R)$ 所确定的 PTS 包括它的句子、初始上下文、判断以及类型指派规则的定义，其中：

1. 句子的形式是：$A:B$，其中 A 和 B 都是初始项。

2. 初始上下文的形式是：$\Gamma=[x_1:A_1,\cdots,x_n:A_n]$，其中 x_i 是变量，A_i 是初始项。

3. 判断的形式是：$\Gamma\vdash M:A$，其中 Γ 是初始上下文，M 和 A 是初始项。

4. 类型指派规则用来生成判断：

axioms $\dfrac{c:s\in A}{\vdash c:s}$

var $\dfrac{\Gamma\vdash A:s\quad x\notin\Gamma}{\Gamma,x:A\vdash x:A}$

weakening $\dfrac{\Gamma\vdash M:A\quad \Gamma\vdash C:s\quad x\notin\Gamma}{\Gamma,x:C\vdash M:A}$

product $\dfrac{\Gamma\vdash A:s_1\quad \Gamma,x:A\vdash B:s_2\quad (s_1,s_2,s_3)\in R}{\Gamma\vdash \Pi x:A.B:s_3}$

application $\dfrac{\Gamma\vdash F:(\Pi x:A.B)\quad \Gamma\vdash M:A}{\Gamma\vdash FM:B[M/x]}$

abstraction $\dfrac{\Gamma,x:A\vdash M:B\quad \Gamma\vdash(\Pi x:A.B):s}{\Gamma\vdash(\lambda x:A.M):(\Pi x:A.B)}$

conversion $\dfrac{\Gamma\vdash M:A\quad \Gamma\vdash B:s\quad A=_\beta B}{\Gamma\vdash M:B}$

纯类型系统的一个重要成果是给出了 λ 方阵，λ 方阵中所有 PTS 规范的类子集是 $\{*,\Box\}$，公理集是 $\{*:\Box\}$，但具有不同的规则集。下面给出 λ 方阵中每个演算的规则集：

1. $\lambda\to$ 演算：$(*,*)$

2. $\lambda 2$ 演算：$(*,*)$ $(\Box,*)$

3. λP 演算：$(*,*)$ $(*,\Box)$

4. $\lambda P2$ 演算：$(*,*)$ $(\Box,*)$ $(*,\Box)$

5. $\lambda\underline{\omega}$ 演算：$(*,*)$ $(\Box,\Box)$

6. $\lambda\omega$ 演算：$(*,*)$ $(\Box,*)$ $(\Box,\Box)$

7. $\lambda P\underline{\omega}$ 演算：$(*,*)$ $(*,\Box)$ $(\Box,\Box)$

8. $\lambda P\omega$ 演算：$(*,*)$ $(\Box,*)$ $(*,\Box)$ $(\Box,\Box)$

用图可表示为图1。而它们所能表达的逻辑系统可对应地用图2表示，其中，PROP 表示直觉命题逻辑，2表示二阶，ω 表示高阶，PRED 表示直觉谓词逻辑，该图对应地被称为 L 方阵。

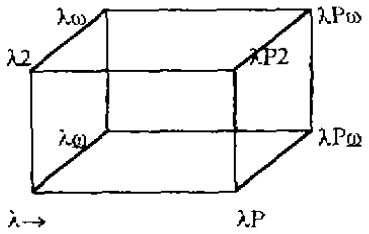

图1

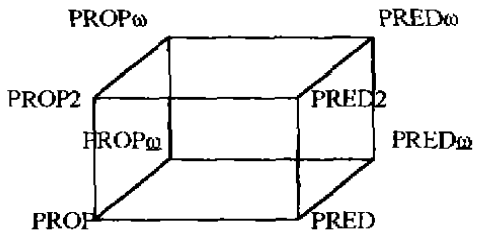

图2

在上面的演算系统中，λP 又称为 LF，即 Edinburgh 的逻辑框架，也是 de Bruijn 的 AUTOMATH 系统中所使用的类型系统。λPω 就是构造演算。

如果说若有规则(s_1, s_2)就称类子 s_2的 Γ 类型的构造依赖于 s_1的 Γ 类型的话，那么看到在 λ 方阵中有下面的依赖关系：

1. 所有的演算中，项的构造依赖于项，例如有 MN 这样的项。

2. λ2演算中，项的构造依赖于类型，例如有 $M\alpha$ 这样的项。

3. $\lambda\underline{\omega}$ 演算中。类型的构造依赖于类型，例如有 αβ 这样的项。$\lambda\underline{\omega}$ 是不含多态性，即没有∀ 量词，但有 Λ 抽象的高阶 λ 演算。

4. λP 类型的构造依赖于项，即所谓的依赖积，即有 $\Pi x{:}A.B$ 这样的类型，其中 A 和 B 是类型。变量 x 代表的是项。且在 B 中含有 x。

其他的系统的依赖性可类推。

研究纯类型系统的一个重要目标是一致地研究各种类型的性质，这些性质包括：类型检查、类型推导和类型下有否项等问题的可判定性；规则的结构性质；约简规则与特定约简下的规范式、强规范性等问题以及主题约简等。关于 PTS 的更详细的讨论及它对于这些性质的一些重要研究成果。读者可参考文[24,26]等。

5 类型系统的应用

类型系统的一个直接应用是用于编译程序的研究与开发，其次是设计新型的程序设计语言。目前设计新型程序语义的研究热点集中于将函数式程序设计语言与面向对象程序设计语言结合起来，著名的有 OML 及 ML2000，它们都是在 ML 语言中引入面向对象的思想，也有设计新的过程式的面向对象程序设计语言，因为 Smalltalk、Eiffel 等语言从某种程度上看，都是类型不安全的，C++是类型安全的，但它对于类型和动态联编等限制得太严格，目的是要设计限制不太严格，而又类型安全的语言。这方面的工作有 K. Bruce 等的 LOOM 语言等。

在 Martin-Lof 提出直觉类型理论之后，许多学者研究它用于程序自动开发和验证。逻辑类型系统可很好地用于程序开发和验证，一方面是因为可将类型看作程序的规范，而项可看作满足该规范的程序，另一方面因为逻辑类型系统将表示程序数据类型的机制和表示程序规范（逻辑公式）的机制结合在一起，从而可在统一的框架下研究程序的自动开发和程序正确性的验证。

对类型系统的应用研究最多的还是定理的自动证明，在 80 年代早期就有 de Bruijn 等人的 AUTOMATH 项目，该项目对推动类型系统研究的发展起了重要作用。现在还在研究和开发的系统有主要基于直觉类型理论的 NuPRL 系统，基于构造演算的 Coq 系统以及基于扩充的构造演算的 LEGO 系统等。将类型系统用于定理的自动证明的基本思想都来源于 Curry-Howard 对应，将类型看成逻辑公式，而将该类型下的项看成是该公式的证明。大多数的类型系统能表示高阶逻辑，因此基于类型系统的定理证明系统的功能很强大。

作者认为对于面向对象类型系统的研究不仅对研究面向对象理论的形式基础，如面向对象程序语言的形式语义有重要的意义，对面向对象软件的开发也有重要的意义。目前软件工程研究中的热点之一是软件体系结构的研究，而面向对象软件的软件体系结构无非是构件与构件之间的关系。将类型是程序的规范，项是满足规范的程序具体化为类型是可重用构件的规范，项是满足规范的可重用构件同样有意义，这对软件体系结构的形式化研究，乃至软件重用技术的形式化研究都有重要的意义。

结束语 本文对类型系统研究的主要方面做了介绍，但并没有也不可能完全覆盖类型系统研究的各个方面。象类型系统的各种扩充所引入的交类型、并类型、商类型、递归类型、积类型、和类型、记录类型等没

有深入讨论，每个演算为了增强它的表达能力，都会引入一些常量。

国内对于类型系统的研究相对来说较少，在80年代中后期，由于 Martin-L of 直觉类型理论的影响，国内开始开展了有关类型系统(理论)的研究，集中在逻辑类型系统研究方面[27]。目前随着面向对象思想的日益重要，国内也开展了有关面向对象程序的类型系统研究[28,29]。总的来说，国内对于类型系统的研究还很少，但随着国内对理论计算机科学研究的发展，类型系统的研究必会引起越来越多的学者注意和研究。

参 考 文 献

1 Girard J. Interpretation fontionelle et élimination des coupures de l' arithmétique d' ordre superieur : [PhD thesis] Universite Paris Ⅶ, 1972
2 Reynolds J. Towards a theory of type structure. In: Proc. Colloque Sur la Programmation. New York, Springer-Verlag LNCS 19. 1974. 408～425
3 Matin-L o f P. Intuitionistic Type Theory. Bibliopolis, Napoli, 1984
4 Nordstr om B, et al. Programming in Martin-L of's Type Theory, An Introduction. Oxford University Press, 1990
5 Cardlli L. A semantics of multiple inheritance. Information and Computation, 1988, 76(2/3): 136～164
6 Cardelli L, Wegner P. On understanding types, data abstraction, and polymorphism. Computing Surveys, 1985, 17(4): 471～522
7 Compagnoni A B. Higher-Order Subtyping with Intersection Types: [PhD Thesis]. University of Edinburg, 1995
8 Castagna G, et al. A calculus for overloaded functions with subtyping. In: ACM Conf. on LISP and Functional Programming. San Francisco, 1992. 182～192
9 Cardelli L, Longo G. A semantic basis for Quest. Journal of Functional Programming, 1991, 1(4): 417～458
10 Cardelli L, et al. An extension of System F with subtyping. Information and Computation, 1994, 109(1-2): 4～56
11 Abadi M, Cardelli L. A Theory of Primitive Objects: Second-order System. In European Symposium on Programming (ESOP), Edinburgh, Scotland, 1994
12 Abadi M, Cardelli L. A Theory of Primitive Objects--Untyped and First-Order Systems. In Theoretical Aspects of Computer Software (TACS). Sendai, Japan, 1994
13 Fisher K, et al. A lambda calculus of objects and method specialization. Nordic Journal of Computing (formerly BIT) 1994, 1: 3～37
14 Fisher K, Mitchell J C. The Development of Type Systems for Object-Oriented Languages. Theory and Practice of Object Systems, 1996, 1/3: 189～220
15 Pierce B C. Type System. Book Manuscript. Available at: http. //www. cis. upenn. edu/～bcpierce/
16 Compagnoni A B. Subject Reduction and Minimal Types for Higher Order Subtyping: [Technical report ECS-LFCS-97-363]. LFCS, University of Edinburgh, 1997
17 Luo Zhaohui. An Extended Calculus of Constructions: [PhD thesis]. Department of Computer Science, University of Edinburgh, June 1990
18 Coquand T, Huet G. The Calculus of Constructions. Information and Computation, 1998, 76(2/3): 95～120
19 Pierce B C. Bounded quantification is undecidable. Information and Computation, 1994, 112(1). 131～165
20 Pierce B C, Steffen M. Higher-order subtyping. Theoretcial Computer Science, 1995
21 Cook W R. A Denotational Semantics of Inheritance. [PhD thesis] Brown University, 1989
22 Pierce B C, Turner D N. Simple typetheoretic foundations for object-oriented programming. Journal of Functional Programming, 1994, 4(2): 207～248
23 Bruce K, et al. Comparing Object Encodings. Information and Computation, 1998
24 Barendregt H. Lambda calculi with types. In Gabbay Abramsky and Maibaum, editors, Handbook of Logic in Computer Science, Volume Ⅰ, Oxford University Press, 1992
25 Barthe G, Hatcliff J. A Notion of Classical Pure Type System (Preliminary version). Electronic Notes in Theoretcial Computer Science, 1997, 6
26 Jacobs B. Categorical Type Thoery: [PhD thesis]. University of Nijmegen, 1991
27 傅育熙. 类型理论原理. 软件学报, 增刊, 1997年6月: 457～465
28 全炳哲, 金淳兆, 玄顺姬. 面向对象类型理论的比较研究. 计算机研究与发展, 1997, 34(10): 736～741
29 全炳哲, 金淳兆, 李文辉. 基于类型理论的面向对象程序设计. 计算机学报, 1997, 20(1): 50～57

(上接第4页)

参 考 文 献

1 Landauer R. IBM J. Research and Development, 3, 183 (1961)
2 Feynman R. Int. J. Theory Physics, 21, 467(1982)
3 Feynman R. Optical News 11, 11 (1985) (reprinted in Found. Phys. 16, 507(1986))
4 Keyes R W. IBM J. Research and Development, 32, 24 (1988)
5 Feynman R, Hibbs A R. Quantum Mechanics and Path Integrals(1965), McGraw-Hill, ISBN 0-07-020650-3
6 Ekert A, et al. Review of Modern Physics, 68, p733(1996)
7 许良英等, 编译. 爱因斯坦文集. 第一卷. 商务印书馆, 1976. 366～367
8 Bell J S. Physics, vol. 1, 195(1964)
9 Aspect A, et al. Physics Review Letter, 49, 1804(1982)
10 Preskill J. Available at: http: // xxx. soton. ac. uk/ps/quant-ph/9705031, 1997