类型理论和程序设计

李 未　　王飓安

(100083北京航空航天大学计算机科学系)

摘　要

Type theory is a formalism which studies the principles of type definitions and deductions. It is currently one of the most active areas in computer science. This paper gives a survey of the origin and development of type theory. The relationships among type theory, logics and algebras are discussed and some distinguished type systems are introduced. We put emphasis on the applications of type theory to programming languages and software development.

一、类型理论的起源与发展

类型理论的基本思想和方法学基础来源于哲学中的归类(classification)方法，即把具有共同特点的个体对象归入一类，并把具有共同特点的类集合成超类的思维过程和方法。归类是形成概念的先决条件之一，分类方法在认识和实践活动中有广泛的应用。例如生物学中的分类范畴典型体现了类型分划思想：以种为单位，相近的种集合为属，相近的属集合为科，科隶于目，目隶于纲，纲隶于门，构成生物分类谱系。在类型理论中，具有相同性质和结构的项的集合形成类型；具有相似性质的类型的集合构成超类型，项隶于类型，类型从属于超类型等等，这种过程也体现了归类的思想方法。

然而，类型理论的直接来源是数学。

十九世纪七十年代，德国数学家Cantor创立集合论，奠定了现代数学的基础。但在1900年前后，人们在Cantor集合论中发现悖论，引起了第三次数学危机。为了消除悖论，摆脱危机，人们提出了各种改造数学的方案，如公理化集合论，函数及其施用的逻辑系统(Church, 1932)等等。英国数学家Russell分析了数学悖论产生的本质原因，认为这些悖论与非直谓定义有关。所谓非直谓定义，是指被定义的对象包括在借以定义它的各种对象之中，例如"一切集合的集合"等。于是Russell提出了消除数学悖论的第三种方案：分支类型理论，其基本思想是对Cantor集合论中的概括原则施加约束条件，从而限制造集的任意性。后来Ramsey把分支类型理论加以简化，得出简单类型理论。

Russell和Ramsey的类型概念和类型分划思想对数学的发展作出了重要贡献。应该说，后来各种带类型的λ-演算系统、程序设计中的类型理论都发源于Russell早年提出的见解，并且借鉴了其中的方法。

λ-演算是三十年代由逻辑学家Church提出的，另外，Berkley, Rosser, Stephen Kleen等人在早期的发展中起了重要作用。组合逻辑在定义计算时不使用变量，为高阶逻辑提供了有效的方法。这种思想首先在二十年代由逻辑学家Moses Schönfinkel提出，

后来Haskell Curry进一步发展了这种方法。λ-演算和组合逻辑最初都是作为高阶逻辑的组成部分而发展起来的。其目的是刻画计算的最原始、最一般的性质，建立关于函数的一般理论，为逻辑和数学提供可靠的基础。尽管这个目标并未完全达到，但是其中关于函数的理论，即λ-演算，却获得了巨大的成功。Church用"λ-可定义性"对"有效可计算"概念进行了形式化描述；A.Turing证明了用λ-演算可以描述所有Turing可计算函数。值得一提的是，第一个关于不可计算性的结果就是在λ-演算中发现和表述的。

在λ-演算中，所有的表达式都看作函数，所有的运算都转化为函数施用，因而无类型λ-演算可以看成只有一个类型：由值到值的函数类型，同时所有的值本身又是具有同样类型的函数。这种无类型λ-演算或称单类型λ-演算存在以下三个问题：第一，表达繁琐，不便应用。比如我们希望某些函数用来表达布尔值和布尔运算，而另一些函数表达整数值和整数运算；这些区别不仅在语用上方便，而且具有方法学上的意义。第二，不能保证每个项都有标准型或称为范式，比如$(\lambda x.xxy)(\lambda x.xxy)$将有无穷归约。即使一个项有标准型，也不能保证它的每一种归约方式都终止，且终止于同样的结果。比如令$L=(\lambda x.xxy)(\lambda x.xxy)$，$P=(\lambda u.v)L$，则P至少有如下两种归约方式：

(i) $P\equiv(\lambda u.v)L\longrightarrow[L/u]v=v$

(ii) $P\rightarrow(\lambda u.v)(Ly)\rightarrow(\lambda u.v)(Lyy)\rightarrow\cdots$

第一种归约方式终止于标准型，而第二种归约方式不终止。

第三，λ-项并不是数学家通常使用的集合论函数的概念。通常我们讲函数时，总把它的定义域和值域看作该函数定义的组成部分，而无类型λ-演算中的项根本不存在定义域和值域的概念。

上述三个问题都可以通过对λ-演算增加类型系统来解决。带类型的λ-演算就是类型理论的雏型。

在带类型λ-演算中，每个项都有唯一的标准型，而且强范式化(strong normalization)定理成立。因而带类型λ-演算是关于全函数的理论。另外，带类型λ-演算是可判定的，并且简单带类型λ-演算具有自然的模型。但是，简单带类型λ-演算表达能力有限，它只能表示原始递归函数的子集。目前在类型理论中的一个研究方向就是在简单带类型λ-演算中增加新的类型构子，扩充类型层次，增加类型系统的表达能力。

类型是程序设计中常用的重要概念。程序设计语言经历了从无类型到带类型的发展过程，而且其类型结构的丰富程度已成为评价该语言优劣的标准。另一方面，类型理论用于软件生产自动化，机器定理证明和人工智能的前景非常诱人。类型可以看作规范，具有此类型的项对应着相应规范的程序实现，类型推导过程就相当于程序的构造和正确性验证。下面我们还要详细说明，类型理论与逻辑、代数、计算机程序设计语言、软件开发都有着密切关系。

二、类型理论与逻辑的关系

在类型理论发展史上，是Curry首先发现，如果在类型指派系统TA_C[2]中把所有的主体去掉，把类型构子"→"解释为蕴含，则得到的结果就是直觉主义命题演算中的纯蕴含部分。后来许多人运用Curry的"公式作为类型"或"命题作为类型"的思想，把这种对应关系推广到其它的联结词和量词。较有影响的工作有de Bruijn[3]，W.A.Howard[4]。目前，文献中常把"公式作为类型"或"命题作为类型"法则称为"Curry-Howard同构"。这种同构对应关系是沟通逻辑与类型理论的桥梁，是目前大多数类型理论的共同基础。下面我们选取八个类型系统来说明类型理论与逻辑之间的对应关系。按照惯例，我们用x表示变元，c表示常元；所有类型组成的类称为超类型(kind)，记为*；所有超类型组成的类记为#。

定义一

表达式 $E::=x\mid c\mid EE'\mid\lambda x:E\cdot E'\mid \pi x:E\cdot E'$

β-归约 $(\lambda x:A\cdot B)C\rightarrow[C/x]B$

对于$A、B\in E$，$A:B$称为类型指派语句，其中A称为主体，B称为谓体。上下文是由主体是相异变元的语句组成的序列，记为Γ。若$\Gamma=\langle x_1:A_1,\ \cdots,\ x_n:A_n\rangle$，则$\Gamma,x:B=\langle x_1:A_1,\ \cdots,\ x_n:A_n,\ x:B\rangle$。类型指派规则的形式为$\Gamma\vdash A:B$，它表示语句$A:B$可由上下文Γ导出。令$s,\ s_1,\ s_2\in S=\{*,\ \#\}$。

一般类型指派规则

R_1 $\langle\ \rangle\vdash *:\#$

R_2 $\dfrac{\Gamma\vdash A:S}{\Gamma,\ x:A\vdash x:A}$，x在Γ中不出现。

R_3 $\dfrac{\Gamma\vdash A:B\quad \Gamma\vdash C:S}{\Gamma,\ x:C\vdash A:B}$

R_4 $\dfrac{\Gamma\vdash F:\pi x:A\cdot B\quad \Gamma\vdash C:A}{\Gamma\vdash FC:[C/x]B}$

R_5 $\dfrac{\Gamma\vdash A:B\quad \Gamma\vdash B':S\quad B=\beta B'}{\Gamma\vdash A:B'}$

特殊类型指派规则(亦称$(s_1,\ s_2)$-规则)

R_6 $\dfrac{\Gamma\vdash A:s_1\quad \Gamma,\ x:A\vdash B:s_2}{\Gamma\vdash(\pi x:A\cdot B):s_2}$

R_7 $\dfrac{\Gamma\vdash A:s_1\quad \Gamma,x:A\vdash b:B\quad \Gamma,x:A\vdash B:s_2}{\Gamma\vdash(\lambda x:A\cdot b):(\pi x:A\cdot B)}$

根据$(s_1,\ s_2)$-规则选取的方式，可得到如下八种类型系统：

T_1：R_1—R_5+$(*,\ \#)$-规则，并且要求$\pi x:A\cdot B$中x不在B中出现。T_1就是Church在1940年提出的带类型λ-演算，它是最简单的类型系统，是其它类型系统的基础。

T_2：R_1—R_5+$(s_1,\ s_2)$-规则，这里$(s_1,\ s_2)\in\{(*,\ *),(\#,*)\}$。$T_2$称为二阶带类型λ-演算或多态λ-演算，它是Girard在1972年提出的F系统[5]的子系统。Reynold在1974年独立地给出了二阶带类型λ-演算。Girard的研究以证明论为出发点，而Reynolds则着重于程序设计理论。目前二阶多态类型系统在程序设计语言和语义理论中有广泛的应用。

T_3：R_1—R_5+$(s_1,\ s_2)$-规则，其中$(s_1,\ s_2)\in\{(*,\ *),\ (*,\ \#)\}$。$T_3$是de Bruijn在1980年提出的AUTOMATH[3]语言族中的一种形式。爱丁堡逻辑框架ELF[7]，ALT类型系统[8]也都是基于T_3的类型理论。但是，ELF采用"判断作为类型"观点，而ALT采用"概念作为类型"的观点。下面将会看到，如果采用"命题作为类型"的Curry-Howard同构原则，则T_3将对应于谓词逻辑。

T_4：R_1—R_5+$(s_1,\ s_2)$-规则，其中$(s_1,\ s_2)\in\{(*,\ *),\ (\#,\ *),\ (*,\ \#)\}$。$T_4$是Longo和Moggi在1988年研究的类型系统λP_2。在[9]中给出了T_4与二阶谓词逻辑的关系。

T_5：R_1—R_5+$(s_1,\ s_2)$-规则，这里$(s_1,\ s_2)\in\{(*,*),(\#,\#)\}$。$T_5$是R. de Lofalette在1985年研究多态类型和递归类型时提出的类型系统。

T_6：R_1—R_5+$(s_1,\ s_2)$-规则，其中$(s_1,\ s_2)\in\{(*,*),\ (\#,*),\ (\#,\#)\}$。$T_6$本质上是Girard[5,6]中的系统$F_\omega$。

T_7：R_1—R_5+$(s_1,\ s_2)$-规则，其中$(s_1,\ s_2)\in\{(*,\ *),\ (*,\ \#),\ (\#,\#)\}$，这个类型系统目前尚未被深入研究。

T_8：R_1—R_5+$(s_1,\ s_2)$-规则，其中$(s_1,\ s_2)\in\{(*,*),(*,\#),(\#,*),(\#,\ \#)\}$。$T_8$是Coquand和Huet在1988年提出的构

类型系统	(s_1,s_2)-规则	对应的逻辑
T_1	$(*,*)$	命题逻辑
T_2	$(*,*),(\#,*)$	二阶命题逻辑
T_3	$(*,*),(*,\#)$	谓词逻辑
T_4	$(*,*),(\#,*),(*,\#)$	二阶谓词逻辑
T_5	$(*,*),(\#,\#)$	弱高阶命题逻辑
T_6	$(*,*),(\#,*),(\#,\#)$	高阶命题逻辑
T_7	$(*,*),(*,\#),(\#,\#)$	弱高阶谓词逻辑
T_8	$(*,*),(*,\#),(\#,*),(\#,\#)$	高阶谓词逻辑

造演算CC[10]的变种。CC是用于构造性证明的高阶类型系统，它可以看成是高阶函数式程序设计语言。

显然，这八个类型系统根据规则集的包含关系可以构成一个代数结构格，它同构于格$\langle \mathcal{P}(s \mid |s| = 3), \subseteq \rangle$。根据Curry-Howard同构原则，上述类型系统与逻辑间的对应关系如上表所示。

三、类型理论与代数的关系

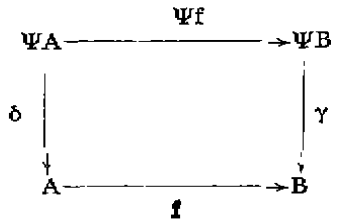
从程序设计的观点看，λ-演算就是抽象的函数式程序设计语言，因为λ-演算具有程序设计语言及其实现的许多特点，比如λ-演算中的约束变元对应过程中的形参，施用运算对应过程调用等等。因此，λ-演算可以作为程序设计语言的语义模型。目前用带类型λ-演算处理程序设计语言及其类型推导和检查已非常流行，并且已取得许多重要成果。

但是从代数学家的观点来看，λ-演算只是纯碎的语法实体，它的语义需要用代数模型进行解释。近几年来代数理论在计算机科学中应用越来越广泛，其中最重要的成果之一是发现类型理论与范畴理论有着密切的关系。下面仅举几个有代表性的工作[5,6,9,13,14,15]为例，其中主要的结论有：无类型λ-演算与C-monoid本质上相同；带类型λ-演算和笛卡尔闭范畴本质上相同；直觉主义类型理论与topos，Martin-Löf的类型理论与局部笛卡尔闭范畴，构造演算与代数topos，都有着密切的关系。下面以几种较简单的类型系统来说明这种对应关系，所使用的术语和方法均采用代数风格，由此我们可以看出代数方法的优美和简洁。

带类型的λ-演算由三部分组成：类型集合T，也称为L的类型系统；项的集合$\{\mathrm{Term}_t \mid t \in T\}$；项之间的等式公理。一般类型系统中有基本类型B，函数类型$t \to t'$，积类型$t \times t'$。多态类型系统要求在基本类型中包含一个类型变元集T_v，即$B = B_1 + T_v$，并且在基本类型中增加全称类型$\forall t.t'$，这里$t \in T_v$，t'是类型。

根据范畴理论，类型和项都是范畴$\mathbf{Se}_t$上某个endo-函子的初始代数，这里Set是所有小集合及小集合上的全函数分别作为对象和态射所构成的范畴。

设C是任意范畴，Ψ是C上的endo-函子，Ψ的不动点是序偶(A, δ)，其中$\delta: \Psi A \to A$是同构。如果δ是任意态射，则称(A, δ)是Ψ-代数。若(A, δ)，(B, γ)都是Ψ-代数，则可定义Ψ-代数的态射f如下：$f: (A, \delta) \to (B, \gamma)$是使下图交换的C-态射$f: A \to B$：



可以证明Ψ-代数及其态射构成范畴Ψ-Alg，它的初始对象是Ψ的不动点，称为最小不动点。

定义二 Ψ_1，Ψ_2是Set→Set的endo-函子：

$$\Psi_1(x) = B + (X \times X)_r + (X \times X)_l$$

$$\Psi_2(x) = B + (X \times X)_s + (X \times X)_l + (T_v \times X)_v$$

于是一般类型系统T_1就是初始Ψ_1-代数，多态类型系统T_2就是初始Ψ_2-代数。

四、类型理论与程序设计语言的关系

计算机程序设计语言与类型理论有密切的关系。λ-演算分为无类型λ-演算和带类型λ-演算，相应地，计算机程序设计语言也可分为两类：无类型语言和带类型语言。在类型理论中，每个数据都看成某种类型的元素，复杂的类型可以从基本类型出发通过诸如笛卡尔积、不交并、函数空间等数学运算来构造。在程序设计语言(例如Pascal)中也有类似的构造，这些语言提供诸如布尔类型、整数类型、字符串类型等原始类型和类

型构子，如数组、记录等，于是由原始类型和类型构子可以构造出复杂的类型。

在程序设计语言发展的早期阶段，程序设计语言中没有类型或只有简单的类型结构。ALGOL60是第一个带类型说明并在编译时进行类型检查的程序设计语言，随后发展起来的PL/1,Pascal, ALGOL68, Simula, Modula-2, Smalltalk, ML, Miranda, Haskell等语言都带有丰富的类型结构。

在类型理论中有两种基本的类型指派方式：隐式类型指派(又称为 Curry指派方式)和显式类型指派(又称为Church指派方式)。在隐式类型指派中，表达式是无类型的λ-项，对每个项可指派多个类型。例如既有$(\lambda x.x):\alpha\to\alpha$，又有$(\lambda x.x):(\alpha\to\beta)\to(\alpha\to\beta)$。在显式类型指派中，表达式的类型由其子表达式的类型唯一确定，例如$(\lambda x:A.x):A\to A$。现有的程序设计语言大多采用隐式类型指派方式，如ML, HOPE, Miranda等。爱丁堡LCF的类型系统采用了显式类型指派方式。

在程序设计语言中加入类型系统的主要目的是指明语言成分的性质，比如数据的范围，表达式的特性，函数的作用域等，从而保证语言的正确使用。一个语言的类型系统包括静态类型结构和类型推导系统两部分，静态类型结构是指用该语言编写程序时允许使用的类型信息和规则，比如怎样为常元、变元、算子和函数指派类型以及指派什么样的类型。如果通过静态分析就可以确定程序中每个表达式的类型，则这样的语言就称为静态类型化语言。静态类型化语言要求程序员为程序中每个变量和表达式都给出显式类型，编译程序时只需对这些显式类型信息进行相容性检查。为了减少程序员的工作量，大多数语言尽量避免完全显式地给出类型信息，比如ML允许程序员只给出部分类型信息甚至略去类型说明，这就需要由类型推导系统在编译时根据上下文确定表达的类型。

显然，程序设计语言中静态类型结构越丰富，它的类型推导系统就越复杂。目前类型理论应用的重要领域之一就是为程序设计语言类型推导系统提供理论基础和可靠方法。

近来面向对象的程序设计方法颇为流行。Cardelli和Wegner[16]把面向对象定义为：

面向对象＝数据抽象＋对象类型＋类型继承

可见面向对象方法与类型理论有着十分密切的关系。目前关于面向对象程序设计中的类型理论研究很活跃，有兴趣的读者可参阅[16,17,18]。

五、类型理论与软件开发的关系

在类型理论中最基本的断言形式是a:A，它有两层含意：第一，A是一个合法的类型；第二，a是类型A的一个项或a是类型A的一个对象。由于在抽象类型理论中类型是不加定义的基本概念，因而对类型作不同的语义解释就得到类型理论的不同应用。比如若把类型A解释为某个逻辑L中的命题公式，a看作A的证明，我们就能用类型理论解释和实现逻辑L；若把类型A解释为软件的规范描述，a看作正确实现规范A的程序，则我们就能用类型理论描述软件的形式化开发以及程序正确性验证。下面分四种情形简要介绍较典型的类型解释和应用。

1.命题作为类型

这种解释起源于Curry, Howard作了进一步发展，并在de Bruijn的AUTOMATH中首先得到实际应用。关于命题与类型之间的深刻对应关系已在[20]中有详细论述，下面只简要介绍几个基于"命题作为类型"原则的类型理论和系统。

1.1 Martin-Löf直觉主义类型理论

这个理论的核心是进行如下四种形式的判断：(1) A是类型，(2) A和B是相等的类型，(3)a是类型A的一个对象，(4)a和b是类型A的相等的对象。其中第三种判断可理解为：a是集合A的一个元素；a是命题A的一

个证明；a是任务A的一个算法。如果把类型A看作规范说明，则程序a(类型A的对象)就是满足规范说明的一种实现。从而Martin-Löf直觉主义类型理论既包含了规范描述语言，又包含了程序设计语言，同时还具有从规范说明到程序实现的形式化推导规则。构造$a \in A$的过程不但由规范导出了程序，而且给出了导出程序满足规范说明的证明，这就使程序规范说明、程序构造和程序验证一体化，为软件的形式化开发提供了统一的框架。用Martin-Löf直觉主义类型理论进行程序开发的实例可参见[20，21，22，23]。

1.2 **Nuprl系统** Nuprl系统由Cornell大学Constable等人开发，现已成为类型理论成功地应用于计算机科学的标志之一。Nuprl系统遵循"命题作为类型"的原则，其基本类型有整数int，串类型atom，空类型void和类型域谱系U_1，U_2，U_3，…，每个类型都属于该谱系的某个域；复杂的类型可以由基本类型出发通过类型构子构造出来。典型的类型构子有函数空间"→"，相关积"×"，不交并"|"，集合，递归等。根据Curry-Howard同构原则，对谓词演算中每个公式f都有一个类型T与之对应，f是有效的当且仅当T不空。根据这种对应，Nuprl充分地利用了逻辑证明中蕴含的计算信息，比如给定一个构造主义存在性断言的证明之后，Nuprl系统可以利用证明中的计算信息抽取出表示该断言为真的对象(程序)来。另外，具有如下形式的断言"对类型为A的任意对象x，可以构造类型为B的对象y满足关系R(x，y)"的证明隐式地定义了一个从A到B的可计算函数f，Nuprl系统能够从证明中构造出这个f来，而且可以对类型为A的输入计算f的值。

从软件开发角度看，Nuprl系统可以作为程序综合器：给定一个有计算意义的断言即程序的规范描述，再给一种证明策略，Nuprl就可以从证明中自动抽取出程序，在证明完成的同时，我们就得到满足规范的正确程序。Nuprl系统还可看成计算机辅助问题求解系统以及实现数学的系统。当然，作为计算机系统，Nuprl还拥有支持文本编辑、证明生成和函数求值的交互式环境并具有智能系统的某些特征[28]。

1.3 **PX系统** 这是由东京大学S·Hayashi开发的基于Curry-Howard同构原则的计算机系统。PX采用"证明作为程序"的观点，其目标是检查构造性证明并从证明中抽取程序。上面介绍的两种系统抽取出来的程序本质上是λ-表达式，而PX抽取出来的是实际的LISP程序。根据类型理论原理，具有合法类型的程序既是终止的(强范式化定理)又正确实现了程序规范，所以用PX开发出来的程序是完全(totally)正确的。PX系统的另一特点是采用了递归(称为CIG-递归)和其它控制构造，因而提高了效率。

1.4 **构造演算** 构造演算(Calculus of Costruction)由法国INRIA的T.Coquand和G.Huet提出，它扩充了de Bruijn的AUTOMATH系统，既包含Girard的高阶λ-演算又含有Martin-Löf类型理论中的相关类型，从而它把相关类型的直谓理论与非直谓的高阶λ-演算融为一体。构造演算是用自然演绎风格进行构造性证明的高阶形式系统，每个证明都是带类型的λ-表达式，根据Curry-Howard原则，证明和断言分别对应于程序和规范，从而构造演算可被看作高层次的函数式程序设计语言，或是通用的泛函式系统。构造演算已在INRIA实现，大量的数学证明可在其中表述和进行机器检查，它的应用与算法开发见[24，25]。

2.判断作为类型

"判断作为类型"是爱丁堡大学ELF类型理论的基本原则。ELF的目标是建立一个关于逻辑系统的一般理论，从而为开发独立于具体逻辑系统的各种软件工具建立可靠的理论基础。用ELF可以定义计算机科学中常见的各种逻辑系统，具体方法分为三步：第一，对逻辑的抽象语法中每种语法范畴都用一个类型表示，每种表达式构子都用一个适当的类

面向对象的可视知识研究

何克清　马江　吕美玲　谢彪　金明源

（430072武汉大学软件工程国家重点实验室）

摘　要

As a research progress, this paper presents a new study on object-oriented visual knowledge. First, we propose some concepts and primary achievements, such as object-oriented visual knowledge, knowledge structure, inheritance, knowledge driven and inference, classification, etc. Then, we discuss the application prospect. Also, we put some problems in further research.

1. 前　言

人们对知识的认识和积累可以说是从可视的"形"开始的。在对图形、图象认识的基础上通过抽象获得相应的知识。当今，大量利用抽象知识的同时，我们不应该忽视可视知识的研究。而且，可视知识的利用十分直观、直接、自然、有效。

软件工程的发展，已形成了自己实用的图式方法与技术，如DFD, Petri网, PAD, FC, SC等知识。如何表示和利用这些可视知识，是软件工程中的重要课题之一。

知识工程所取得的方法与技术为可视知

型常元表示。第二，建立对象逻辑中项（公式）与第一步得到的类型的有效项之间的一一对应。第三，用"判断作为类型"的原则解释对象逻辑中的推导规则和证明。这样，ELF就为解释和实现各类逻辑系统提供了统一的框架和方法。

3.概念作为类型

"概念作为类型"是李未在ALT类型理论中提出的原则。在软件工程和知识工程中有许多概念都是成对出现的，比如规范和程序，任务和解答，问题和算法，命题和证明等等，如果用类型理论中类型及其对象的定义来刻画这些概念，就可以利用类型理论中的方法来解决软件工程与知识工程中的各种问题。目前，ALT类型理论已用于描述变换式程序设计，为形式化描述软件开发过程打下了基础。

近来国外也出现了类似的工作，比如程序开发演算[28]和软件开发的类型模型OROS[27]。

六、结　论

类型理论和构造性数学都起源于对证明论和数学基础的研究，近来在计算机科学中引起了广泛注意[20]。类型理论已成为函数式程序设计的逻辑基础[23]，与面向对象程序设计有密切的关系[18]；它不仅对证明检查、自动定理证明有重要意义，而且可作为人工智能中的知识表示语言，它与逻辑、范畴论、语义理论相结合，为软件开发的形式化与自动化开辟了诱人的前景。（参考文献共29篇略）。