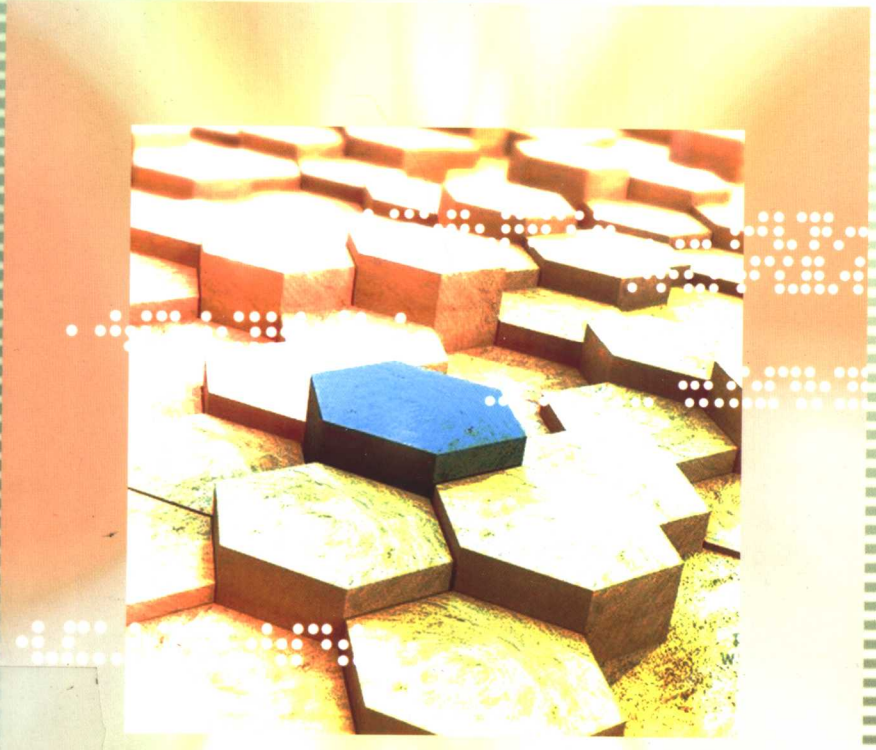
高等学校研究生系列教材

程序设计语言理论

Theory of Programming Languages

陈意云



高等教育出版社
HIGHER EDUCATION PRESS

高等学校研究生系列教材

程序设计语言理论

陈意云

高等教育出版社

内容提要

本书给出分析程序设计语言语法性质、操作性质和语义性质的一个框架，该框架基于λ演算系统。全书围绕着λ演算的一个序列来组织，该序列中λ演算的类型系统依次变得越来越复杂，这些λ演算用来分析和讨论相应的程序设计语言概念，如多态性、抽象数据类型、子类型等。以类型系统为中心对程序设计语言进行的这种研究，在软件工程、语言设计、高性能编译器、计算机和网络安全等方面有着重要应用。

本书可作为高等院校计算机科学及相关专业的研究生教材，也可供计算机软件工程高级技术人员参考。

图书在版编目(CIP)数据

程序设计语言理论/陈意云.—北京：高等教育出版社，2004.9
ISBN 7-04-015516-8

Ⅰ.程… Ⅱ.陈… Ⅲ.程序语言-研究生-教材 Ⅳ.TP312

中国版本图书馆 CIP 数据核字(2004)第 095055 号

策划编辑 刘建元　**责任编辑** 孙惠丽　**市场策划** 陈　振
封面设计 王凌波　**责任印制** 孔　源

出版发行	高等教育出版社	**购书热线**	010-64054588
社　　址	北京市西城区德外大街 4 号	**免费咨询**	800-810-0598
邮政编码	100011	**网　　址**	http://www.hep.edu.cn
总　　机	010-58581000		http://www.hep.com.cn
经　　销	新华书店北京发行所		
印　　刷	北京铭成印刷有限公司		
开　　本	787×1092　1/16	**版　　次**	2004 年 9 月第 1 版
印　　张	22	**印　　次**	2004 年 9 月第 1 次印刷
字　　数	450 000	**定　　价**	32.00 元

本书如有缺页、倒页、脱页等质量问题，请到所购图书销售部门联系调换。

版权所有　侵权必究

物料号：15516-00

前　言

本书给出了分析程序设计语言语法性质、操作性质和语义性质的一个框架，该框架基于一个叫做λ演算的数学系统。λ演算的主要特征是一种用于函数和其他可计算值的表示法以及一种计算表达式的等式逻辑和规则。本书围绕着λ演算的一个序列来组织，该序列中λ演算的类型系统依次变得越来越复杂。这些λ演算用来分析和讨论相应的程序设计语言概念。本书强调顺序语言，但是其中许多概念和技术可以应用到并行程序设计语言。

类型系统根据各程序短语计算的值的种类来对程序短语进行分类，它是一种自动地检查某些错误行为不会出现的一种易驾驭的语法方法。采用类型论观点的程序设计语言类型系统的研究，在软件工程、语言设计、高性能编译器、计算机和网络安全等方面有着重要应用。

本书的λ演算序列有下面几个部分:

（1）最简单的是一个等式系统，又叫做泛代数。这个逻辑没有函数变量，它可以用来公理化和分析程序设计中使用的普通数据类型。

（2）第二个系统是带函数类型（还可以包括笛卡儿积类型以及可区分并类型）的λ演算系统。

（3）在（2）的基础上增加递归定义后，所形成的系统提供了研究函数式语言的操作性质和语义性质的一个有用框架。当把（1）的代数数据类型组合进来后，它对定义许多常规的程序设计语言是足够的。特别是，定义了存储单元类型和状态类型后，它就可以用来研究命令式语言传统的公理语义、操作语义和指称语义。

（4）介绍一些类型论的概念，它们是程序设计语言多态性和数据抽象的基础；研究有多态类型（还带有抽象数据类型和程序模块的声明形式）的λ演算系统。

（5）最后一个λ演算系统是带子定型的λ演算系统。子定型是类型上的一种关系，它隐含一个类型的值可以代替另一个类型的值。

（6）本书最后讨论一些λ演算系统的类型推断算法。类型推断是把“无类型的”或“部分类型化的”项变换成“良类型”项的一般问题，它通过推导未给出的类型信息来完成这个变换。

本书是为计算机软件和理论专业的研究生编写的，其内容主要是根据参考文献 1 整理的，它可以作为进行程序设计语言和类型论方面高级研究的技术参考，也可以作为掌握程序设计语言理论的关键概念的学习资料。具备程序设计语言和形式逻辑的基本知识是学习

本书的必要条件。

中国科学院软件研究所研究员柳欣欣先生审阅了全书，并提出了许多宝贵的意见，在此表示衷心的感谢。

由于作者水平有限，书中难免存在一些缺点和错误，恳请广大读者批评指正。

作者
中国科学技术大学
2004年4月

此为试读，需要完整PDF请访问：www.ertongbook.com

目　录

第1章 引 言

1.1 基本概念

1.1.1 模型语言

对程序设计语言进行数学分析通常是从设计模型语言开始，而且模型语言的设计一般要突出感兴趣的程序构造，忽略一些无关的细节。例如，若想分析过程调用机制，可先设计一个简单的程序设计语言，它的主要构造是过程声明和调用，然后分析该简化语言。这种方法不仅对分析现行的程序设计语言有用，而且有助于新语言的设计。因为实际的程序设计语言是非常庞大和复杂的，语言的设计必须仔细地和分离地考虑各个子语言。当然，必须记住，从简化语言可能会得出一些错误印象。因此，在进行程序设计语言的理论分析和把理论分析用于实际情况时，不要忘记所做的简化及这些简化对结论的影响。

20 世纪 60 年代，人们发现，可以把一个复杂的程序设计语言形式化为两部分：一部分是能抓住该语言本质机制的一个非常小的核心演算；另一部分是一组导出形式，它们的行为可以通过把它们翻译成这个核心演算来理解。通过这种方式来理解语言要深刻得多。这个核心语言就是λ演算（Lambda Calculus）。

λ演算是 20 世纪 20 年代确立的一种形式系统，它源于可计算理论，是奠定函数定义和程序设计语言中命名约定的基本机制。在这种系统中，所有的计算都归约到函数定义和函数应用这样的基本操作。20 世纪 60 年代以来，λ演算已广泛用于规范程序设计语言的特征、语言设计和实现、和类型系统的研究中。它的重要性在于，它可以同时被看成一种简单的程序设计语言（用于描述计算）和一个数学对象（有关它的一些严格的陈述可以证明）。

本书将用类型化λ演算（Typed Lambda Calculus）的框架来研究程序设计语言的各种概念。用不同方式来扩充基本的类型化λ演算，可以设计出多种模型语言，它们包含历史的、现代的，甚至将来的语言特征。盯在类型化λ演算上的一个优点是，所建立的理论有某种程度的“模块性”。类型化λ演算的许多扩充可以组合在一起，一般来说，这种组合不会出现出乎意料的叠加影响（当然也会有例外）。例如，在分别调查了多态性和记录后，很容易定义出含多态性和记录的语言，并且指出它的很多性质。

本书研究程序设计语言的概念和特征的目的是透过表面的语法，对各种程序短语（表达式、命令和声明等）理解到一个适当详细的程度。本章介绍一个非常简单的、以自然数和布尔值作为基本类型的、基于类型化λ演算的语言，介绍该语言的语法、操作语义和它在程序设计中的能力。该语言是第2章介绍的可计算函数程序设计（Programming Computable Functions，PCF）语言的一个简化版本。通过这个简要介绍，读者可以对本书将要采用的技术和方法有一个浅显的了解。

本章的主要议题如下：

(1) λ表示法和λ演算系统概述；

(2) 类型和类型系统的扼要讨论；

(3) 基于表达式的归纳、基于证明的归纳和良基归纳法。

本书以后各章也按此风格，即每一章有一节导言，导言中包含该章的主要议题。

1.1.2　λ表示法

在描述、分析和实现程序设计语言时，λ演算已被证明是非常有用的。稍加练习，读者很快就会熟悉这种表示法，并且可以明白，C、Pascal 和 Ada 的程序短语都是在λ表达式的基础上做了语法美化。这就使得本书描述的理论更加有用，而且使得程序设计语言的多样性更容易理解。语言 PCF 将直接使用λ表示法，就像 Lisp 语言那样；所不同的是，PCF 没有表和原子，但它有相对严格的编译时的定类规则。

λ表示法的主要特征是**λ抽象**和**λ应用**，前者用于定义函数，后者允许使用定义的函数。用λ表示法写出的表达式叫做**λ表达式**或**λ项**。λ表示法既可用于类型化λ演算，也可用于无类型λ演算。

在类型化λ演算中，函数的论域（Domain）由给出形式参数的类型来指定。如果变元 x 的类型是 σ，M 是基于这个假定的合式（Well Formed）表达式，那么 $\lambda x{:}\sigma.\, M$ 定义一个函数，它把 σ 类型的任何 x 映射到由 M 给定的一个值。一个简单的λ抽象的例子是

$$\lambda x : nat.\, x$$

它是自然数上的恒等函数。记号 $x : nat$ 说明该函数的论域是 *nat*，即自然数类型。在该表达式中，点后面的部分是函数体。因为该例的函数体也是 x，因此该函数的值域也是 *nat*。

每一种形式的类型化λ演算，都有精确的规则来说明，在一个给定的变量类型的假设下，什么样的表达式是合式的。这些规则表明怎样从函数体 M 的形式去确定 $\lambda x : \sigma.\, M$ 的值域。例如，表达式 $\lambda x : nat.\, x + \text{true}$ 不是合式的，因为 *true* 和自然数相加是没有意义的。

用程序设计语言定义恒等函数，最熟悉的形式可能是

$$Id\,(x : nat) = x$$

但是这种形式迫使为每个函数起一个名字，而λ表示法给出了直接定义函数的一种简洁方式。例如：

此为试读，需要完整PDF请访问：www.ertongbook.com

$$\lambda x : nat.\ x + 1$$

定义了自然数上的后继函数。而

$$\lambda x : nat.\ 10$$

是自然数上的一个常函数，该函数的值恒为 10。

在λ抽象中，λ是一个约束算子，这意味着在λ项$\lambda x : \sigma.\ M$中，变元 x 是一个占位符，就像在谓词演算公式

$$\forall x : A.\ \phi$$

中的变元 x 那样。因此可以重新命名λ约束变元而不改变表达式的含义，只要所选择的新变元与其他已经使用的变元没有冲突便可以了。仅仅约束变元名字不同的项称为α等价。如果 M 和 N 是α等价的，可以写成 $M =_\alpha N$。如果 x 出现在表达式 M 中，并且它出现在形式为$\lambda x{:}\sigma.\ N$的子表达式中，那么称 x 的这个出现是**约束的**（Bound），否则是**自由的**。注意，在一个表达式中，x 可能既有约束出现也有自由出现。不含自由变元的表达式称为**闭表达式**。

在λ表示法中，用项的并置来表示函数应用，并且用括号来说明运算对象的结合。例如，把恒等函数应用于 5，可写成

$$(\lambda x : nat.\ x)\ 5$$

这个函数应用的结果是 5，即

$$(\lambda x : nat.\ x)\ 5 = 5$$

下一节将看到，可以有几种不同的方式来计算λ表达式的值或证明λ表达式间的等式。

λ表示法中有两个约定，以省略λ表达式中的大量括号。第一个约定是函数应用是左结合的，即 MNP 应看成$(MN)P$；第二个约定是每个λ的约束范围尽可能地大，一直到表达式的结束或碰到不能配对的右括号为止。例如，$\lambda x{:}\sigma.\ MN$解释为$\lambda x{:}\sigma.\ (MN)$，而不是$(\lambda x{:}\sigma.\ M)N$。同样地，$\lambda x{:}\sigma.\ \lambda y{:}\tau.\ MN$ 是$\lambda x{:}\sigma.\ (\lambda y{:}\tau.\ (MN))$的简写。这两个约定可以一起使用。例如，多元函数应用可写成

$$(\lambda x{:}\sigma.\ \lambda y{:}\tau.\ \lambda z{:}\rho.\ M)NPQ$$

它是

$$(((\lambda x{:}\sigma.\ (\lambda y{:}\tau.\ (\lambda z{:}\rho.\ M)))N)P)Q$$

的简写。

1.1.3 记号和约定

本书用到的数学基础主要是集合论的知识，包括关系和函数。这些是大家已经熟悉的，本书不再重复。本书用到的各种归纳法另用一节专门介绍。

本书使用几种形式的相等符号，这些符号及它们的含义如下：

$=$ 两个表达式有相同的值；

$\equiv$ 除了约束变元的名字可能不同以外，两个表达式语法上相同；

$\overset{def}{=}$　用 $M \overset{def}{=} N$ 来表示符号或表达式 M 被定义成等于 N；

::=　在文法中用来表示表达式的可能形式；

≅　表示（集合、代数等的）同构。

本书使用的逻辑符号如下：

∀　全称量词。公式$\forall x.\ \phi$可以读成“对所有的 x，ϕ为真”；

∃　存在量词。公式$\exists x.\ \phi$可以读成“存在一个 x 使得ϕ为真”；

∧　合取。公式$\phi \wedge \psi$ 可以读成“ϕ 合取ψ”；

∨　析取。公式$\phi \vee \psi$ 可以读成“ϕ 析取ψ”；

¬　否定。公式$\neg\phi$ 可以读成“非ϕ”；

⊃　蕴涵。公式$\phi \supset \psi$ 可以读成“ϕ 蕴涵ψ”（不使用→作为蕴涵，因为→用于类型表达式，并用于表达式的归约（计算）；

iff　当且仅当。

本书中使用的集合运算符号如下：

∈　属于；

∪　并集；

∩　交集；

⊆　子集；

×　笛卡儿积。

1.2　等式、归约和语义

在历史上，λ表示法是**λ演算**的一部分，λ演算是关于λ表达式的一个推理系统。除了语法外，这个形式系统有三个主要部分。按照现代程序设计语言的术语，它们分别叫做**公理语义**、**操作语义**和**指称语义**。逻辑学家可能把前二者叫做证明系统，而把第三者叫做一种模型论。公理语义是推导表达式之间等式的一个形式系统；操作语义是一种基于一个方向的等式推理，叫做**归约**，按计算机科学术语，归约可看成**符号计算**的一种形式；指称语义，或模型论，本质上类似于其他逻辑系统（如等词逻辑或一阶逻辑）的模型论。一个模型是一组集合，每种类型一个集合，这个集合就是对应类型的解释域，并且每个良类型（无类型错误）的表达式都可以解释为相应集合上的一个元素。

本节是对本书常用的语义方法做一个扼要的介绍，让读者有一个粗浅的认识。

1.2.1 公理语义

公理语义用逻辑系统来描述程序的性质，即它是一个证明系统，可用来推导程序及其组成部分的性质。这些性质可以是项之间的等式、给定输入下有关程序输出的断言或其他性质。

现在这里给出的是一个等式公理系统，它有约束变元改名公理，还有把函数应用联系到**代换**的一个公理。为了表示这些公理，需要使用记号$[N/x]M$，它表示 M 中的自由变元 x 用表达式 N 代换的结果。代换时需要注意的是不能把 N 中的自由变元变成约束的。把 M 中的自由变元 x 用 N 代换的最简单办法是，首先将 M 中所有的约束变元改名，使得它们和 N 中的自由变元都有区别，然后再将 x 的自由出现用 N 代替。第 2 章将给出更详细的定义。使用代换，约束变元改名公理可写成

$$\lambda x{:}\sigma.\ M = \lambda y{:}\sigma.\ [y/x]M, \qquad M \text{中无自由出现的} y \qquad (\alpha)$$

例如，$\lambda x{:}\sigma.\ x = \lambda y{:}\sigma.\ y$。

因为项$\lambda x{:}\sigma.\ M$定义了一个函数，因此可以通过用 N 代替 x 来计算该函数应用于 N 的结果。例如，将函数$\lambda x{:}nat.\ x+4$应用于 4 的结果是

$$(\lambda x{:}nat.\ x+4)\ 4 = [4/x](x+4) = 4 + 4$$

更一般而言，等式公理

$$(\lambda x{:}\sigma.\ M)N = [N/x]M \qquad (\beta)_{eq}$$

被称为β 等价公理。本质上，β 等价是说，计算函数应用就是在函数体中用实在变元代替形式变元。除了这些公理和个别其他公理外，等式系统还包含对称性规则、传递性规则和同余规则。同余规则是说，相等的函数作用于相等的变元产生相等的结果，可以写成

$$\frac{M_1 = M_2, N_1 = N_2}{M_1 N_1 = M_2 N_2}$$

当然，为了完全精确，还应说明它们的类型，以保证每个项都有意义。和其他逻辑证明系统一样，类型化λ演算的等式证明规则允许推导任何一组等式前提的逻辑推论。

1.2.2 操作语义

语言的操作语义可用不同的方式给出，一种接近实际实现的方式是定义一个抽象机。它是一种理论上的计算机，然后通过一系列的机器状态变换来计算程序。操作语义最实际的表示就是语言的解释器。操作语义比较抽象的表示是演绎出最终结果的证明系统，或者说是通过一系列步骤变换一个表达式的证明系统。本书集中于操作语义的后一种形式，即定义完备的或逐步计算的证明系统。

前面所列的等式公理的单向形式给出了λ演算的归约规则。直观上讲，基本归约规则描述了单步符号计算，它们可以反复用于表达式的计算，直至得到表达式的最简形式，

或因无最简形式而计算不终止。符号计算过程给出了λ演算的计算特征。因为归约是非对称的，箭头→通常用于一步归约，双箭头↠用于任意步（包括零步）归约。

最核心的归约规则是$(\beta)_{eq}$的单向形式，叫做**β 归约**，写成

$$(\lambda x{:}\sigma.\ M)N \to_{\beta} [N/x]M \qquad (\beta)_{red}$$

因为在代换时约束变元可能需要改名，因此归约是定义在α等价上的。即β 归约的结果依赖于新约束变元的选择，它不是惟一确定的，但是归约产生的任何两个项仅在约束变元的名字上有区别，因此是α等价的。

除了$(\beta)_{eq}$规则外，还有一些其他的归约规则。完整的归约系统及其性质在后面章节讨论。

1.2.3　指称语义

用指称方法定义语言语义的基本思想是：先确定指称物，然后给出语言成分到指称物的语义映射，这个映射要满足：

（1）每个成分都有对应的指称物；

（2）复合成分的指称只依赖于它的子成分的指称。

在类型化λ演算的指称语义中，每个类型表达式对应到一个集合，称为该类型的值集。类型σ 的项解释为其值集上的一个元素。类型$\sigma\to\tau$ 的值集是函数集合，因此项$\lambda x{:}\sigma.M$解释为一个数学函数。纯类型化λ演算的语义是简单的，而它的各种扩充的语义可能会比较复杂。具有挑战性的特征有函数的递归定义、类型的递归定义和多态函数等。除了不讨论递归定义的类型的指称语义外，其他两个概念的指称语义在后面都会详细讨论。

对于熟悉无类型λ演算的读者来说，值得一提的是，无类型λ演算可以从类型化λ演算中派生出来。事实上，考虑无类型λ演算的语义的最自然方式之一是从类型化λ演算的语义开始的。基于这个原因，类型化λ演算被看成是更加基本的系统，更适于作为研究的起点。

1.3　类型和类型系统

在计算机科学和数理逻辑的一般文献中，单词“类型”在不同的上下文中有不同的含义。在本书中，类型是一个基本术语，它的含义由类型系统的精确定义给出。在任何类型系统中，类型提供了所有可能值的全体的一种分类：一个类型是一群有某些公共性质的值。对于不同的类型系统，类型的多少和值所属的类型可能不同。

本节扼要介绍类型系统的有关概念和应用。

1.3.1 类型和类型系统

一个程序变量在程序执行期间的值可以设想为有一个范围，这个范围的一个界叫做该变量的类型。例如，类型 *bool* 的变量 *x* 在程序每次运行时的值只能是该类型的值。如果 *x* 有类型 *bool*，那么布尔表达式 *not* (*x*)在程序每次运行时都有意义。变量都被给定（非平凡）类型的语言叫做**类型语言**。

语言若不限制变量值的范围，则被称做**未类型化的语言**（Untyped Language）。它们没有类型，或者说仅有一个包含所有值的泛类型。在这些语言中，一个运算可以作用到任意的运算对象，其结果可能是一个有意义的值、一个错误、一个异常或一个未做说明的结果。

类型语言的**类型系统**是该语言的一个组成部分，它始终监视着程序中变量的类型，通常还包括所有表达式的类型。一个类型系统主要由一组**定型规则**（Typing Rule）构成，这组规则用来给各种语言构造（程序、语句、表达式等）指派类型。

在计算机科学中，类型系统的研究有两个分支：比较实际的一支是类型系统在程序设计语言中的应用；比较抽象的一支关心“纯类型化λ演算”和不同逻辑之间的对应关系。本书主要研究前面一个分支。

为语言设计类型系统的目的是什么？简单讲，**类型系统**的根本目的是用来证明程序不会出现不良行为，例如将整数和布尔值相加。只有那些顺从类型系统的程序才被认为是类型语言的真正程序，其他的程序在它们运行前都应该被抛弃。

当用一种程序设计语言所能表示的所有程序运行时都没有不良行为出现时，就说该语言是**类型可靠的**（Type Sound），类型可靠的语言又称为安全语言。显然，希望通过适当的分析来对程序设计语言的类型可靠性做出准确的回答。这种分析将基于语言的类型系统，这样，类型系统的研究也需要形式化的方法。

类型系统的形式化需要研究精确的表示法和定义，需要一些形式性质的详细证明，这是本书的一个重要内容。形式化的类型系统提供了概念上的工具，用它们可以评判语言定义的一些重要方面的适当性。非形式语言的描述通常不能把语言的类型结构详细规范到不会出现歧义实现的程度，因此经常会出现同一个语言的不同编译器实现了略有区别的类型系统。而且，许多语言定义被发现不是类型可靠的，甚至经过**类型检查**（Type Check）后接受的程序也会崩溃。理想的状况是，形式化的类型系统应该是类型化程序设计语言的定义的一部分。这样，依靠精确的规范，证明语言是类型可靠的才有可能。

一种语言由一个实质存在的类型系统来类型化，而不在乎类型是否实际出现在程序的语法中。一种语言，如果类型是语法的一部分，那么该语言就是**显式类型化的**；否则便是**隐式类型化的**。主流语言中不存在纯隐式类型化的语言，但是 ML 和 Haskell 这样的语言支持编写忽略类型信息的程序片段，这些语言的类型系统会自动地给这些程序片段指派类型。

1.3.2　类型语言的优点

从工程的观点看，类型语言有下面一些优点：

（1）开发时的实惠

有了类型系统可以较早地发现错误，例如整数和串相加。若类型系统很好地设计，类型检查可以发现大部分日常的程序设计错误。因为大部分错误已经被排除，剩下的错误也就很容易调试。

对于大规模的软件开发来说，接口和模块有方法学上的优点，类型信息在这里可以组织到程序模块的接口中。程序员可以一起讨论要实现的接口，然后分头编写要实现的对应代码。这些代码之间的相互依赖最小，并且代码可以局部地重新安排而不用担心对全局造成影响。

程序中的类型信息还具有文档作用。程序员声明标识符和表达式的类型，也就是告诉了所期望的值的部分信息，这对阅读程序是很有用的。

（2）编译时的实惠

程序模块可以相互独立地编译，例如 Modula-2 和 Ada 的模块，每个模块仅依赖于其他模块的接口。这样，大系统的编译可以更有效，因为改变一个模块并不会引起其他模块的重新编译，至少在接口稳定的情况下是这样。

（3）运行时的实惠

在编译时收集类型信息，保证了在编译时就能知道数据所占空间的大小，因而可得到更有效的空间安排和访问方式，提高了目标代码的运行效率。例如，像 Pascal 的记录、C 的结构和对象，其域或成员的偏移可以根据它们的类型信息静态地确定。

另外，一般来说，精确的类型信息在编译时可以保证运行时的运算都作用到适当的对象并且不需要昂贵的运行时的测试，从而提高程序运行的效率。例如在 ML 中，精确的类型信息可以删除在指针脱引用（Dereference）中的 nil 检查。

上面提到，类型信息具有文档作用，但是它和其他形式的程序评注不同。一般来说，关于程序行为的评注可以从非形式的注解一直到用于定理证明的形式规范。类型处在该范围的中间：它们比程序注解精确，比形式规范容易理解。另外，类型系统应该是透明的：程序员应该能够很容易预言一个程序是否可通过类型检查，如果它不能通过类型检查，那么其原因应该是明显的。

除了这些传统的应用外，在计算机科学和有关学科中，类型系统现在还有许多应用，这里列举其中一些。

（1）类型系统一个越来越重要的应用领域是计算机和网络安全。在网络计算中出现的安全问题，像移动代理的资源访问、信任管理，也能够通过类型系统来解决。例如，编译时收集的类型信息附加在移动代码中，可供移动代码接受方检查该代码是否符合基本安全

此为试读，需要完整PDF请访问：www.ertongbook.com