类型系统

卢卡·卡德利

微软研究

1 引言

类型系统的基本目的是在程序运行过程中防止出现执行错误。 这个非正式的陈述激发了对类型系统的研究，但需要进一步澄清。 它的准确性首先取决于一个相当微妙的问题，即什么构成了执行错误，我们将详细讨论。 即使这个问题解决了，没有执行错误是一个非平凡的属性。 当这样的属性对于在编程语言中可以表达的所有程序运行都成立时，我们说这种语言是类型安全的。 事实证明，需要进行相当仔细的分析才能避免对编程语言的类型安全性做出错误和尴尬的声明。 因此，类型系统的分类、描述和研究已经成为一个正式的学科。

类型系统的形式化需要精确的符号和定义的发展，以及对形式属性的详细证明，从而对定义的适当性产生信心。 有时候这门学科变得相当抽象。 然而，人们应该始终记住，基本动机是实用的：这些抽象是出于必要性，并且通常可以直接与具体直觉相关联。 此外，形式技术不需要完全应用才能有用和有影响力。了解类型系统的主要原则可以帮助避免明显和不太明显的陷阱，并且可以激发语言设计中的规律性和正交性。

当正确开发时，类型系统提供了用于评判语言定义中重要方面的概念工具。 非正式的语言描述通常无法足够详细地指定语言的类型结构，以允许无歧义的实现。 经常发生的情况是，同一语言的不同编译器实现略有不同的类型系统。 此外，许多语言定义被发现是类型不安全的，即使程序通过类型检查器被判断为可接受，也可能导致程序崩溃。 理想情况下，形式化的类型系统应该是所有有类型的编程语言定义的一部分。 这样，类型检查算法可以根据精确的规范进行明确的衡量，并且如果可能和可行的话，整个语言可以被证明是类型安全的。

在这个介绍部分中，我们提供了一种非正式的类型命名法，用于类型、执行错误和相关概念。 我们讨论了类型系统的预期属性和好处，并且我们回顾了类型系统的形式化方法。 介绍中使用的术语并不完全标准；这是由于标准术语的固有不一致性引起的。 一般来说，我们在涉及运行时概念时避免使用诸如“类型”和“类型检查”之类的词语；例如，我们用动态检查替代动态类型，并避免使用常见但模糊的术语，如强类型。 这些术语在定义术语部分中进行了总结。

在第2节中，我们解释了常用于描述类型系统的符号表示法。 我们回顾了关于程序类型的形式化断言（判断）、类型规则（判断之间的蕴含关系）和推导（基于类型规则的推理）。
在第3节中，我们回顾了一系列简单类型，这些类型在常见的编程语言中都有对应的类似类型，并详细介绍了它们的类型规则。 在第4节中，我们介绍了一个简单但完整的命令式语言的类型规则。 在第5节中，我们讨论了一些高级类型构造的类型规则：多态和数据抽象。 在第6节中，我们解释了如何通过子类型的概念扩展类型系统。 第7节是对我们略过的一些重要主题的简要评论。 在第8节中，我们讨论了类型推断问题，并介绍了我们考虑的主要类型系统的类型推断算法。 最后，第9节是对成就和未来方向的总结。

执行错误

执行错误最明显的症状是出现意外的软件故障，例如非法指令故障或非法内存引用故障。

然而，还有更微妙的执行错误，导致数据损坏，没有任何直接的症状。 此外，还有一些软件错误，例如除以零和取消引用空指针，这些错误通常不会被类型系统阻止。 最后，有一些没有类型系统的语言，尽管如此，软件错误也不会发生。 因此，我们需要仔细定义我们的术语，从什么是类型开始。

有类型和无类型的语言

程序变量在程序执行过程中可以假定一系列的值。 这个范围的上界被称为变量的类型。 例如，类型为布尔的变量x在程序的每次运行中只能假定布尔值。 如果x的类型是布尔的，那么布尔表达式not(x)在程序的每次运行中都有一个合理的含义。 变量可以被赋予（非平凡的）类型的语言被称为有类型的语言。

不限制变量范围的语言被称为无类型语言：它们没有类型，或者等效地说，具有包含所有值的单一通用类型。 在这些语言中，操作可能应用于不适当的参数：结果可能是一个固定的任意值、一个错误、一个异常或一个未指定的效果。 纯 λ-演算是无类型语言的极端情况，从不发生错误：唯一的操作是函数应用，由于所有值都是函数，该操作永远不会失败。

类型系统是类型化语言的组成部分，它跟踪变量的类型，以及程序中所有表达式的类型。 类型系统用于确定程序是否行为良好（后续讨论）。 只有符合类型系统的程序源代码才应被视为类型化语言的真正程序；其他源代码在运行之前应被丢弃。

一种语言之所以有类型，是因为它存在一个类型系统，无论类型是否实际出现在程序的语法中。 如果类型是语法的一部分，则被称为显式类型的语言；否则被称为隐式类型的语言。 没有主流语言是纯粹的隐式类型的，但像ML和Haskell这样的语言支持编写大型程序片段的方式。

类型信息被省略；这些语言的类型系统会自动为这样的程序片段分配类型。

执行错误和安全性

有必要区分两种执行错误：一种会立即导致计算停止，另一种在一段时间内不被察觉，然后导致任意行为。 前者被称为被捕获错误，而后者被称为未被捕获错误。

一个未捕获错误的例子是错误地访问合法地址，例如，在没有运行时边界检查的情况下访问数组末尾之后的数据。 另一个可能在任意长度的时间内不被注意到的未捕获错误是跳转到错误的地址：那里的内存可能或可能不代表指令流。 被捕获的错误的例子包括除以零和访问非法地址：计算立即停止（在许多计算机体系结构上）。

如果程序片段不会导致未捕获的错误发生，则其是安全的。 所有程序片段都是安全的语言称为安全语言。 因此，安全语言排除了最隐蔽的执行错误形式：可能不被注意到的错误。 非类型化语言可以通过执行时检查来强制执行安全性。 类型化语言可以通过静态地拒绝所有潜在不安全的程序来强制执行安全性。 类型化语言还可以使用运行时和静态检查的混合。

尽管安全性是程序的关键属性，但很少有类型化语言专门关注消除未捕获错误。 类型化语言通常旨在排除大量的陷阱错误和未捕获错误。 我们接下来讨论这些问题。

执行错误和良好行为的程序

对于任何给定的语言，我们可以将可能的执行错误的子集指定为禁止错误。 禁止错误应包括所有未捕获错误以及部分陷阱错误。 如果程序片段不会导致任何禁止错误发生，则称其具有良好行为，或者等效地说是良好行为的。 （相反的是具有不良行为，或者等效地说是不良行为的。）特别地，良好行为的片段是安全的。 所有（合法的）程序都具有良好行为的语言称为强制检查的语言。

因此，对于给定的类型系统，对于一个强类型检查的语言，以下内容成立：

- 不会发生未捕获的错误（安全保证）。
- 不会发生被指定为禁止错误的捕获错误。
- 可能会发生其他捕获错误；程序员有责任避免它们。

通过在编译时执行静态检查，类型化语言可以强制执行良好的行为（包括安全性），以防止不安全和不良行为的程序运行。 这些语言是静态检查的；检查过程称为类型检查，执行此检查的算法称为类型检查器。 通过类型检查器的程序被称为类型良好的；否则，它是类型不良的，这可能意味着它实际上是不良行为的，

或者仅仅是无法保证它是良好行为的。 静态检查的语言的例子包括ML、Java和Pascal（注意Pascal具有一些不安全的特性）。

未类型化的语言可以通过执行足够详细的运行时检查来实施良好的行为（包括安全性），以排除所有禁止的错误。 （例如，它们可以检查所有数组边界和所有除法操作，在禁止的错误发生时生成可恢复的异常。）这些语言中的检查过程称为动态检查；LISP是这种语言的一个例子。 即使这些语言既没有静态检查，也没有类型系统，它们仍然是强类型检查的。

即使是静态检查的语言通常也需要在运行时执行测试以实现安全性。例如，通常需要动态测试数组边界。 语言进行静态检查并不意味着执行可以完全盲目进行。

一些语言利用它们的静态类型结构来执行复杂的动态测试。 例如，Simula67的INSPECT，Modula-3的TYPECASE和Java的instanceof构造根据对象的运行时类型进行区分。 尽管这些语言仍然（稍微不正确地）被认为是静态检查的，但部分原因是动态类型测试是基于静态类型系统定义的。 也就是说，用于类型相等性的动态测试与类型检查器在编译时确定类型相等性的算法是兼容的。

缺乏安全性

根据我们的定义，一个行为良好的程序是安全的。 安全性是一个更原始、也许更重要的属性，比良好的行为更重要。 类型系统的主要目标是通过在所有程序运行中排除所有未捕获的错误来确保语言的安全性。 然而，大多数类型系统的设计目标是确保更一般的良好行为属性，并隐含地确保安全性。 因此，类型系统的声明目标通常是通过区分良好类型和非良好类型的程序来确保所有程序的良好行为。

实际上，某些静态检查的语言并不能确保安全性。 也就是说，它们禁止的错误集合并不包括所有未捕获的错误。 这些语言可以委婉地称为弱检查（或弱类型，在文献中），意味着某些不安全的操作在静态检查时被检测到，而其他一些操作则未被检测到。 这个类别的语言在其弱点方面差异很大。 例如，只有在使用未标记的变体类型和函数参数时，Pascal才是不安全的，而C具有许多不安全且广泛使用的特性，例如指针算术和类型转换。 有趣的是，C程序员的前五个十诫[30]是针对C的弱检查方面进行补偿。C中弱检查引起的一些问题在C++中得到了缓解，而Java中更多的问题得到了解决，这证实了远离弱检查的趋势。 Modula-3支持不安全的特性，但只允许在明确标记为不安全的模块中使用，并防止安全模块导入不安全接口。

大多数无类型语言是完全安全的（例如，LISP）。否则，在没有编译时和运行时检查来防止损坏的情况下，编程将会非常令人沮丧。 汇编语言属于无类型不安全语言的不愉快类别。

表1. 安全性

	类型化的	无类型的
安全的	ML, Java	LISP
不安全的	C	汇编语言

编程语言应该是安全的吗?

一些语言，如C语言，故意不安全，因为考虑到性能问题：为了实现安全性，运行时检查有时被认为太昂贵。 即使在进行广泛的静态分析的语言中，安全性也是有代价的：例如，数组边界检查等测试通常无法在编译时完全消除。

尽管如此，已经有许多努力设计C语言的安全子集，并通过引入各种（相对昂贵的）运行时检查来安全地执行C程序的开发工具。 这些努力是由于两个主要原因：C语言在非常重要的性能关键应用中的广泛使用，以及由不安全的C程序引入的安全问题。 安全问题包括由指针算术或缺乏数组边界检查引起的缓冲区溢出和下溢，这可能导致任意内存区域的覆盖，并可被利用进行攻击。

根据不同的度量标准，安全性是具有成本效益的，而不仅仅是纯性能。 在执行错误的情况下，安全性会产生故障停止行为，从而减少调试时间。 安全性保证运行时结构的完整性，从而实现垃圾回收。 反过来，垃圾回收大大减少了代码大小和代码开发时间，但以一定的性能为代价。 最后，安全性已成为系统安全的必要基础，特别是对于加载和运行外部代码的系统（如操作系统内核和Web浏览器）。 系统安全正成为程序开发和维护中最昂贵的方面之一，而安全性可以降低这些成本。

因此，选择安全还是不安全的语言可能最终与开发和维护时间以及执行时间之间的权衡有关。 尽管安全语言已经存在了很多年，但仅仅是最近由于安全问题而变得流行起来。

编程语言是否应该有类型?

关于编程语言是否应该具有类型的问题仍然存在一些争议。
毫无疑问，用非类型语言编写的生产代码只能以极大的困难来维护。 从可维护性的角度来看，即使是弱检查的不安全语言也优于安全但无类型的语言（例如，C与LISP）。 以下是从工程角度提出的支持类型语言的论点：

- 执行经济性。 类型信息最初是为了改善数值计算的代码生成和运行时效率而引入的，例如在FORTRAN中。在ML中，准确的类型信息消除了对指针解引用的*nil*-检查的需要。 总的来说，编译时准确的类型信息可以在运行时应用适当的操作，而无需进行昂贵的测试。

- 小规模开发的经济性。 当类型系统设计良好时，类型检查可以捕捉到大部分常规编程错误，消除冗长的调试会话。 发生的错误更容易调试，因为排除了大量其他错误。此外，有经验的程序员采用一种编码风格，使一些逻辑错误显示为类型检查错误：他们将类型检查器作为开发工具。 （例如，当字段的不变性发生变化时，即使其类型保持不变，也更改其名称，以便在所有旧用法上报告错误。）

- 编译的经济性. 类型信息可以组织成接口，用于程序模块，例如M*o*du*la*-2和Ad*a*。然后可以独立编译每个模块，每个模块仅依赖于其他模块的接口。

  在大型系统的编译中更加高效，因为至少在接口稳定时，对一个模块的更改不会导致其他模块重新编译。

- 大规模开发的经济性. 接口和模块在代码开发中具有方法论优势。 大型程序员团队可以协商要实现的接口，然后分别实现相应的代码片段。 代码片段之间的依赖关系最小化，可以在不担心全局影响的情况下进行局部重排。 （这些好处也可以通过非正式的接口规范实现，但在实践中，类型检查对于验证对规范的遵守非常有帮助。）

- 在安全领域的开发和维护经济性. 虽然安全性是消除诸如缓冲区溢出之类的安全漏洞的必要条件，但类型化是消除其他灾难性安全漏洞的必要条件。 这是一个典型的例子：如果有任何方法，无论多么复杂，将整数转换为指针类型（或对象类型），那么整个系统都会受到威胁。 如果可能的话，攻击者可以根据所选择的类型访问系统中的任何数据，即使在一个有类型的语言中，也可以在其中查看数据。 另一个有用的（但不是必要的）技术是将给定的类型化指针转换为整数，然后再转换为上述不同类型的指针。 在消除这些安全问题方面，从维护和可能也是整体执行效率的角度来看，最具成本效益的方法是使用类型化语言。 尽管如此，安全问题在系统的各个层面上都存在：类型化语言是一个很好的基础，但并不是一个完整的解决方案。

- 语言特性的经济性. 类型构造自然以正交方式组合。 例如，在P*ascal*中，数组的数组模拟二维数组；在ML中，一个带有单个参数的过程，该参数是一个元组的参数模拟了一个带有参数的过程。 因此，类型系统促进了语言特性的正交性，质疑了人为限制的效用，从而减少了编程语言的复杂性。

类型系统的预期属性

在本章的其余部分，我们假设语言应该既安全又类型化，因此应该使用类型系统。 在类型系统的研究中，我们不区分被困和未被困的错误，也不区分安全性和良好行为：我们专注于良好行为，并将安全性视为一种隐含的属性。

在编程语言中，类型通常具有实用特性，这使它们与其他类型的程序注释区分开来。 一般来说，关于程序行为的注释可以从非正式注释到受定理证明的形式规范。 类型位于这个光谱的中间：它们比程序注释更精确，比形式规范更容易机械化。 以下是任何类型系统所期望的基本属性：

- 类型系统应该是可判定的：应该有一个算法（称为类型检查算法），可以确保程序的行为良好。 类型系统的目的不仅仅是陈述程序员的意图，而是在错误发生之前积极捕捉执行错误。 （任意的形式化规范没有这些属性。）

- 类型系统应该是透明的：程序员应该能够轻松预测一个程序是否能通过类型检查。 如果不能通过类型检查，失败的原因应该是显而易见的。 （自动定理证明没有这些属性。）
- 类型系统应该是可强制执行的：类型声明应该尽可能地进行静态检查，否则进行动态检查。 应该经常验证类型声明与其关联程序之间的一致性。 （程序注释和约定没有这些属性。）

类型系统的形式化方式

正如我们所讨论的，类型系统用于定义良好类型的概念，它本身是对良好行为（包括安全性）的静态近似。 安全性通过故障停止行为促进调试，并通过保护运行时结构来实现垃圾回收。 良好的类型进一步促进程序开发，可以在运行时之前捕捉执行错误。

但是我们如何保证类型正确的程序真的是良好行为的呢？ 也就是说，我们如何确保语言的类型规则不会意外地允许不良行为的程序通过呢？

形式化类型系统是对编程语言手册中描述的非正式类型系统的数学表征。 一旦类型系统被形式化，我们可以尝试证明一个类型完备性定理，该定理说明类型正确的程序是良好行为的。
如果这样的完备性定理成立，我们称该类型系统是完备的。 （一个类型化语言的所有程序的良好行为和其类型系统的完备性意味着同一件事。）为了形式化一个类型
系统并证明一个完备性定理，我们必须基本上形式化所讨论的整个语言，现在我们简要概述一下。
形式化一个编程语言的第一步是描述其语法。 对于大多数感兴趣的语言来说，这归结为描述类型和术语的语法。类型表达静态

关于程序的知识，术语（语句、表达式和其他程序片段）表达了算法行为。

下一步是定义语言的作用域规则，将标识符的出现与其绑定位置（标识符声明的位置）明确地关联起来。 类型化语言所需的作用域无疑是静态的，即在运行时之前必须确定标识符的绑定位置。 绑定位置通常可以仅通过语言的语法来确定，无需进一步分析；这种静态作用域被称为词法作用域。 缺乏静态作用域被称为动态作用域。

作用域可以通过定义程序片段的自由变量集合（涉及变量如何由声明绑定）来进行形式化规定。 然后可以定义类型或术语替换自由变量的关联概念。

当这些都确定下来后，就可以继续定义语言的类型规则。 这些规则描述了形如 M:A的关系，其中 M是术语， A是类型。 一些语言还需要形如 $A<:B$的类型子类型关系，以及形如 $A=B$的类型等价关系。 语言的类型规则集合构成了其类型系统。具有类型系统的语言称为类型化语言。

在没有引入另一个基本要素之前，类型规则无法形式化，而这个要素在语言的语法中没有反映出来：静态类型环境。 在处理程序片段期间，这些用于记录自由变量的类型；它们与编译器的符号表在类型检查阶段密切对应。 类型规则始终是针对正在进行类型检查的片段的静态环境来制定的。

例如，具有类型关系 M:A的关联静态类型环境 Γ包含有关 M和 A的自由变量的信息。 该关系的完整写法为 $\Gamma \vdash M$:A，表示在环境 Γ中， M具有类型 A。

形式化语言的最后一步是将其语义定义为项和一组结果之间的关系has-*v*alue。 这种关系的形式取决于所采用的语义风格。 无论如何，语言的语义和类型系统是相互关联的：一个术语及其结果的类型应该是相同的（或适当相关的）；这是完备性定理的本质。

类型系统的基本概念适用于几乎所有的计算范式（函数式、命令式、并发等）。个别类型规则通常可以无需修改地应用于不同的范式。 例如，对于函数的基本类型规则，无论语义是按名调用还是按值调用，或者是函数式的还是命令式的，都是相同的。

在本章中，我们独立地讨论类型系统。 然而，应该明白，最终类型系统必须与语义相关，并且该语义应该满足完备性。 可以说，结构操作语义的技术统一处理大量的编程范式，并且与本章的内容非常契合。

类型等价性

如上所述，大多数复杂的类型系统需要定义一种关系，即类型等价性。 在定义编程语言时，这是一个重要问题：当

分别写的类型表达式是否等价？ 例如，考虑两个不同的类型名称，它们与相似的类型相关联：

类型 X =布尔类型
type Y =布尔类型

如果类型名称 X和 Y通过与相似类型相关联而匹配，我们有结构等价。 如果它们由于不同的类型名称而无法匹配（不考虑相关类型），我们有按名称等价。

在实践中，大多数语言使用结构等价和按名称等价的混合。
纯结构等价可以通过类型规则轻松而准确地定义，而按名称等价则更难确定，并且通常具有算法风格。 当需要在网络上存储或传输类型化数据时，结构等价具有独特的优势；相比之下，按名称等价无法轻松处理在时间或空间上分开开发和编译的相互作用程序。

在接下来的内容中，我们假设结构等价（尽管这个问题并不经常出现）。
可以在结构等价中满意地模拟按名称等价，如Modula-3的品牌机制所示。

2 类型系统的语言

类型系统独立于特定的类型检查算法，指定了编程语言的类型规则。 这类似于通过形式语法来描述编程语言的语法，独立于特定的解析算法。

将类型系统与类型检查算法分离是方便且有用的：类型系统属于语言定义，而算法属于编译器。 通过类型系统来解释语言的类型方面要比给定编译器使用的算法更容易。 此外，不同的编译器可能对相同的类型系统使用不同的类型检查算法。

作为一个小问题，技术上可以定义只允许不可行的类型检查算法或根本没有算法的类型系统。然而，通常的意图是允许高效的类型检查算法。

判断

类型系统由一种特定的形式主义描述，我们现在介绍一下。 类型系统的描述从一组称为判断的形式化表达开始。
一个典型的判断形式为：

$\Gamma \vdash \Im$ $\Im$是一个断言； $\Im$的自由变量在 Γ中声明。

我们说 Γ蕴含 $\Im$。 这里 Γ是一个静态类型环境；例如，一个有序列表，包含不同的变量和它们的类型，形式为 $\emptyset, x_1{:}A_1, ..., x_n{:}A_n$。 空环境表示为 $\emptyset$，变量的集合为 $x_1...$在 Γ中声明的变量 x_n由 $dom(\Gamma)$表示，即 Γ的主体。 断言 $\Im$的形式因判断而异，但 $\Im$的所有自由变量必须在 Γ中声明。

对于我们目前的目的来说，最重要的判断是类型判断，它断言一个术语M在自由变量的静态类型环境中具有类型A。它的形式如下：

$\Gamma \vdash M : A$　　M在Γ中具有类型A

示例。

$\emptyset \vdash true : Bool$　　$true$具有类型$Bool$

$\emptyset, x{:}Nat \vdash x+1 : Nat$　　$x+1$具有类型Nat，前提是x具有类型Nat

通常需要其他判断形式；一个常见的判断形式只是断言一个环境是良好形式的：

$\Gamma \vdash \diamond$　　Γ是良好构造的（即已正确构建）

任何给定的判断可以被视为有效的（例如，Γ □ true : *Bool*）或无效的（例如，Γ □ true : *N*at）。 有效性形式化了良好类型的程序的概念。 有效和无效判断之间的区别可以用多种方式表达；然而，一种高度风格化的表达有效判断的方式已经出现。 这种基于类型规则的表达方式有助于陈述和证明关于类型系统的技术引理和定理。 此外，类型规则具有高度模块化的特点：不同构造的规则可以分别编写（与单体类型检查算法相反）。 因此，类型规则相对容易阅读和理解。

类型规则

类型规则基于已知为有效的其他判断来断定某些判断的有效性。 这个过程通过一些本质上有效的判断开始（通常是指空环境是良好形成的判断）。

$$\frac{\begin{array}{l}\text{（规则名称）（注释）}\\ \Gamma_n \mathsf{\Gamma n}_n \text{（注释）}\end{array}}{\Gamma \vdash \Im}$$

类型规则的一般形式。

每个类型规则都写在水平线上方的一些前提判断$\Gamma_i \Im_i$之上，下方是一个单一的结论判断$\Gamma\Im$。 当所有前提都满足时，结论必须成立；前提的数量可以为零。每个规则都有一个名称。 （按照惯例，名称的第一个单词由结论判断决定；例如，“（Val ...）”形式的规则名称用于结论是值类型判断的规则。）在需要时，限制规则适用性的条件以及规则内使用的缩写都会在规则名称或前提旁边进行注释。

例如，以下两个规则中的第一个规则说明任何数字都是一个类型为Nat的表达式，在任何良好构造的环境Γ中。 第二个规则说明两个表示自然数的表达式M和N可以组合成一个更大的表达式$M + N$，该表达式也表示一个自然数。 此外，用于M和N的环境Γ，它声明了M和N的任何自由变量的类型，也适用于$M + N$。

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash n : \text{自然数}}\ (\text{Val } n)\ (n = 0, 1, ...) \qquad \frac{\Gamma \vdash M : \text{自然数} \quad \Gamma \vdash N : \text{自然数}}{\Gamma \vdash M+N : \text{自然数}}\ (\text{Val } +)$$

一个基本规则表明，空环境是良好形成的，没有任何假设:

$$\frac{}{\emptyset \vdash \diamond}\ (\text{环境 } \emptyset)$$

一组类型规则被称为(形式)类型系统。 从技术上讲，类型系统适用于形式证明系统的一般框架：用于进行逐步推导的规则集合。 类型系统中进行的推导涉及程序的类型。

类型推导

在给定的类型系统中，推导是一个带有顶部叶子和底部根的判断树，其中每个判断是通过系统中的某个规则从其上方的判断获得的。 对类型系统的一个基本要求是必须能够检查推导是否正确构造。

一个有效的判断是在给定类型系统中可以作为推导的根获得的判断。 也就是说，有效的判断是通过正确应用类型规则获得的判断。
例如，使用先前给出的三个规则，我们可以构建以下推导，从而证明 □ □1+2 : *Nat*是一个有效的判断。 每一步应用的规则显示在每个结论的右侧：

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \diamond}\ \text{通过 (环境 } \emptyset)}{\square\ \square 1 : \text{自然数}}\ \text{通过 (Val } n) \qquad \dfrac{\dfrac{}{\emptyset \vdash \diamond}\ \text{通过 (环境 } \emptyset)}{\square\ \square 2 : \text{自然数}}\ \text{通过 (Val } n)}{\square\ \square 1+2 : \text{自然数}}\ \text{通过 (Val +)}$$

良好的打字和类型推断

在给定的类型系统中，如果存在类型 *A*，使得环境 Γ下的项 *M*是良好类型的，则 Γ ⊢ *M* : *A*是一个有效的判断；也就是说，项 *M*可以被赋予某种类型。

对于一个项的推导（以及类型的推导），被称为类型推断问题。 在由规则（Env ∅）、（Val *n*）和（Val +）组成的简单类型系统中，在空环境下可以为项1+2推断出一个类型。 这个类型是 *Nat*，在前面的推导中得到的。

假设我们现在添加一个带有前提 Γ □ 和结论 Γ ⊢ *true* : *Bool*的类型规则。 在结果类型系统中，我们无法为项1+*true*推断出任何类型，因为没有规则可以将自然数与布尔值相加。 由于没有任何关于1+*true*的推导，我们说1+*true*是不可类型化的，或者说它是类型错误的，或者说它有一个类型错误。

我们还可以添加一个类型规则，前提是 $\Gamma \vdash M$:自然数和 $\Gamma \vdash N$:布尔值，并且结论是 $\Gamma \vdash M+N$:自然数（例如，将 *true*解释为1）。 在这样的类型系统中，可以为术语 1+*true*推断出一个类型，现在它是良好类型的。

因此，对于给定术语的类型推断问题非常依赖于所讨论的类型系统。 类型推断的算法可能非常简单、非常困难，或者根本无法找到，这取决于类型系统。 如果找到了，最好的算法可能非常高效，或者完全无法使用。 尽管类型系统通常以抽象方式表达和设计，但它们的实际效用取决于良好的类型推断算法的可用性。

对于显式类型的过程性语言（如Pascal），类型推断问题相对容易解决；我们在第8节中进行了讨论。 对于隐式类型的语言（如ML），类型推断问题要复杂得多，我们在这里不进行讨论。 基本算法已经得到了很好的理解（文献中有几种描述），并且被广泛使用。 然而，在实践中使用的算法版本非常复杂，仍在研究中。

在存在多态性的情况下，类型推断问题变得特别困难（在第5节中讨论）。 对于Ada、CLU和Standard ML的显式类型多态特性，类型推断问题在实践中是可处理的。 然而，这些问题通常通过算法来解决，而不是首先描述相关的类型系统。 多态性的最纯粹和最一般的类型系统体现在第5节中讨论的λ-演算中。

这个多态 λ-演算的类型推断算法相当简单，我们在第8节中介绍它。 然而，解决方案的简洁性取决于冗长的类型注释。 为了使这种通用多态性实用化，必须省略一些类型信息。这样的类型推断问题仍然是一个活跃研究领域。

类型的完备性

我们现在已经建立了关于类型系统的所有一般概念，我们可以开始研究特定的类型系统。 从第3节开始，我们将回顾一些非常强大但相对理论的类型系统。 这个想法是通过首先理解这些几个系统，更容易为编程语言中可能遇到的各种复杂特性编写类型规则。

当我们沉浸在类型规则中时，我们应该记住，一个合理的类型系统不仅仅是一组任意的规则。 良好的类型化意味着对良好程序行为的语义概念的对应。 习惯上，通过证明类型完备性定理来检查类型系统的内部一致性.这就是类型系统与语义相遇的地方。

对于指示性语义，我们期望如果 $\vdash M : A$是有效的，那么 $[\![M]\!] \in [\![A]\!]$holds（ M的值属于类型 A所表示的值的集合），对于操作语义，我们期望如果 $\vdash M : A$并且 M减少到 M'那么 $\vdash M' : A$。 在这两种情况下，类型完备性定理断言良好类型的程序在计算时不会出现执行错误。 参见[11, 34]以了解技术概述和最新的完备性证明。

第三章 一阶类型系统

大多数常见过程式语言中的类型系统被称为一阶。 在类型理论术语中，这意味着它们缺乏类型参数化和类型抽象，

这是二阶特性。 一阶类型系统包括（相当令人困惑的）高阶函数。 Pascal和Algol68具有丰富的一阶类型系统，而FORTRAN和Algol60则具有非常差的类型系统。

对于无类型 λ-演算，可以给出一个最小的一阶类型系统，其中无类型 λ-抽象 $\lambda x.M$ 表示一个参数为 x和结果为M的函数。 这个演算的类型系统只需要函数类型和一些基本类型；我们将在后面看到如何添加其他常见的类型结构。

一阶有类型 λ-演算被称为系统F_1。 与无类型 λ-演算相比，主要的变化是为 λ-抽象添加类型注释，使用语法 $\lambda x{:}A.M$，其中x是函数的参数，A是它的类型，而M是函数的主体。 （在有类型的编程语言中，我们可能会包括结果的类型，但在这里不是必需的。）从 $\lambda x.M$到 $\lambda x{:}\,A.M$的步骤是从无类型语言到有类型语言的典型进展：绑定变量获得类型注释。

由于F_1主要基于函数值，最有趣的类型是函数类型：
$A\to B$是具有类型为A的参数和类型为B的结果的函数的类型。 然而，为了开始，我们还需要一些基本类型来构建函数类型。 我们用 *Basic*表示这些类型的集合，并用 $K\in Basic$ 表示任何这样的类型。 在这一点上，基本类型纯粹是一种技术上的必要性，但很快我们将考虑一些有趣的基本类型，如 *Bool*和 *Nat*。

F_1的语法见表2。 简要评论一下语法在类型化语言中的作用是很重要的。 对于无类型的 λ-演算，无上下文的语法描述了合法的程序。 在类型化的演算中，情况并非如此，因为良好的行为不是（通常）上下文无关的属性。 描述合法程序的任务由类型系统接管。例如， $\lambda x{:}K.x(y)$ 符合表2中给出的F_1的语法，但不是F_1的程序，因为它的类型不正确，因为 K不是函数类型。 无上下文的语法仍然是必需的，但仅用于定义自由变量和约束变量的概念；也就是说，用于定义语言的作用域规则。 基于作用域规则，只有在约束变量不同的情况下，例如 $\lambda x{:}K.x$和 $\lambda y{:}K.y$，被认为是语法上相同的。 这种方便的认同在类型规则中被隐式地假设（为了应用某些类型规则，可能需要重新命名约束变量）。

表2. F_1的语法

A,B ::=		类型
K	$K\in$基本	基本类型
$A\to B$		函数类型
M,N ::=		项
x		变量
$\lambda x{:}A.M$		函数
$M\ N$		应用

对于F_1的自由变量的定义与无类型 λ-演算相同，只需忽略类型注释。

对于F_1，我们只需要三个简单的判断，它们在表3中显示。判断 $\Gamma \vdash$ 在某种意义上，A是多余的，因为所有语法上正确的类型A在任何环境 Γ中都自动成立。然而，在二阶系统中，类型的成立性不仅仅由语法来捕捉，判断 $\Gamma \vdash A$变得必要。现在采用这个判断是为了方便后续的扩展。

表3. F_1的判断

$\Gamma \vdash \diamond$	Γ是一个良好形成的环境
$\Gamma \vdash A$	A是在 Γ中形成良好的类型
$\Gamma \vdash M : A$	M是一个类型为 A的良好形式术语在 Γ

这些判断的有效性由表4中的规则定义。规则 (Env $\emptyset$) 是唯一一个不需要假设的规则（即它是唯一的公理）。它声明空环境是一个有效的环境。规则 (Env x) 用于将环境 Γ扩展到更长的环境 $\Gamma, x{:}A$，前提是 A在 Γ中是一个有效的类型。请注意，假设 $\Gamma \vdash A$暗示着 Γ是有效的（归纳地）。也就是说，在推导 $\Gamma \vdash A$的过程中，我们必须已经推导出 $\Gamma \vdash \diamond$。这个规则的另一个要求是变量 x不能在 Γ中被定义。我们在环境中小心地保持变量的不同，这样当 $\Gamma, x{:}A \vdash M : B$被推导出时，就像（Val Fun）的假设中一样，我们知道 x不能出现在 $dom(\Gamma)$中。

表4. F_1的规则

(环境 $\emptyset$)
$$\frac{}{\emptyset \vdash \diamond}$$

(环境 x)
$$\frac{\Gamma \vdash A \quad x \notin \text{属于}(\Gamma)\text{的定义域}}{\Gamma, x{:}A \vdash \diamond}$$

(类型常量)
$$\frac{\Gamma \vdash \diamond \quad K \in \text{基本}}{\Gamma \vdash K}$$

(类型箭头)
$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \to B}$$

(值 x)
$$\frac{\Gamma', x{:}A, \Gamma'' \vdash \diamond}{\Gamma', x{:}A, \Gamma'' \vdash x{:}A}$$

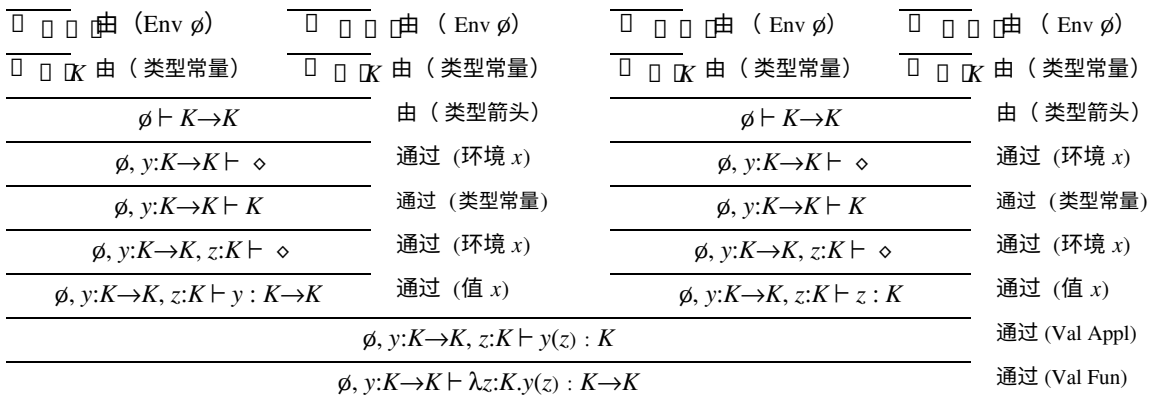
(值函数)
$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : A \to B}$$

(值应用)
$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}$$

规则（类型常量）和（类型箭头）构建类型。规则（Val x）从环境中提取一个假设：我们使用 $\Gamma', x{:}A, \Gamma''$的符号表示，而不是正式地表示 $x{:}A$出现在环境中的某个地方。规则（Val Fun）给予函数类型 $A \to B$，前提是函数体在假设形参具有类型 A的情况下接收类型 B。注意在这个规则中环境的长度如何改变。规则（Val Appl）将函数应用于参数：在验证前提时，相同的类型 A必须出现两次。

表5展示了一个相当大的推导，其中使用了F_1的所有规则。

表5. F_1中的推导



现在我们已经检查了简单的一阶类型系统的基本结构，我们可以开始丰富它，使其更接近实际编程语言的类型结构。 我们将为每个新的类型构造添加一组规则，遵循一个相当规律的模式。

我们从一些基本数据类型开始：类型 *Unit*，它的唯一值是常量 *unit*；类型 *Bool*，它的值是 *true*和 *false*；以及类型 *Nat*，它的值是自然数。

*Unit*类型经常被用作填充不感兴趣的参数和结果；在一些语言中被称为*Void*或 *Null*。 *U*nit上没有操作，所以我们只需要一个规则说明 *Unit*是一个合法的类型，并且一个规则说明 *unit*是 *Unit*类型的一个合法值（表6）。

表6. Unit类型

(类型 单位)	(值 单位)
$\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \text{单位}}$	$\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \text{单位} : \text{单位}}$

对于 *Bool*，我们有类似的规则模式，但布尔值还有一个有用的操作，条件语句，它有自己的类型规则（表7）。 在规则（Val Cond）中，条件语句的两个分支必须具有相同的类型 *A*，因为任何一个分支都可能产生结果。

规则（Val Cond）展示了关于类型检查所需的类型信息量的一个微妙问题。 当遇到条件表达式时，类型检查器必须分别推断 N_1和 N_2的类型，然后找到一个与两者兼容的单一类型 *A*。 在某些类型系统中，可能很难或不可能从 N_1和 N_2的类型中确定这个单一类型。 为了解决这个潜在的类型检查困难，我们使用下标类型来表示额外的类型信息：如果$_A$是类型检查器的一个提示，表示结果类型应该是

A，并且应该分别将推断出的类型 N_1和 N_2与给定的 *A*进行比较。 一般来说，我们使用下标类型来表示在整个考虑的类型系统中可能有用或必要的信息。 类型检查器通常的任务是合成这些额外的信息。 如果可能的话，可以省略下标。 （大多数常见的语言不需要注释 if_A。）

表7. 布尔类型

(布尔类型) $\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash_{\text{布尔类型}}}$　(值为真) $\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \text{真} : \text{布尔类型}}$　(值为假) $\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash \text{假} : \text{布尔类型}}$

(值为条件) $$\frac{\Gamma \vdash M : \text{布尔类型} \quad \Gamma \vdash N_1 : A \quad \Gamma \vdash N_2 : A}{\Gamma \vdash (\mathit{if}_A\ M\ \mathit{then}\ N_1\ \mathit{else}\ N_2) : A}$$

自然数的类型, *N*at (表8), 有0和su*cc* (后继)作为生成元。 或者，就像我们之前所做的那样，一个单一的规则可以说明所有数值常量都具有类型*N*at。对*N*at的计算是通过pred (前驱)和is*Z*ero (零测试)原语实现的；可以选择其他一组原语。

表8. 自然类型

(自然数类型) $\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash_{\text{自然数类型}}}$　(值为零) $\dfrac{\Gamma \vdash \diamond}{\Gamma \vdash 0 : \text{自然数类型}}$　(值为后继) $\dfrac{\Gamma \vdash M : \text{自然数}}{\Gamma \vdash \mathit{succ}M : \text{自然数类型}}$

(值为前驱) $\dfrac{\Gamma \vdash M : \text{自然数}}{\Gamma \vdash \mathit{pred}M : \text{自然数类型}}$　(值为零判断) $\dfrac{\Gamma \vdash M : \text{自然数}}{\Gamma \vdash \mathit{isZero}\ \ M : \text{布尔类型}}$

现在我们有了一系列基本类型，我们可以开始研究结构化类型了，从乘积类型开始(表9)。 乘积类型 $A_1 \times A_2$是具有一对值的类型，第一个分量的类型是 A_1，第二个分量的类型是 A_2。 这些分量可以通过投影 *first*和 *second*来提取。 除了投影之外（或者附加投影），还可以使用 *with*语句来分解一个对 *M*进行绑定，将其分量分别绑定到两个独立的变量 x_1和 x_2在作用域 *N*中。 与ML中的模式匹配相关的 *with*表示法，也与Pascal的 *with*相关；后者与前者的联系将在我们考虑记录类型时变得更加清晰。

乘积类型可以轻松推广为元组类型$A_1 \times ... \times A_n$，具有相应的推广投影和推广 *with*。

表格9. 产品类型

(乘积类型) $\dfrac{\Gamma \vdash A_1 \quad \Gamma \vdash A_2}{\Gamma \vdash A_1 \times A_2}$　(值为对) $\dfrac{\Gamma \vdash M_1 : A_1 \quad \Gamma \vdash M_2 : A_2}{\Gamma \vdash\ M_1, M_2\ : A_1 \times A_2}$

(Val 第一)
$$\frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \text{第一}\ M : A_1}$$

(Val 第二)
$$\frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \text{第二}\ M : A_2}$$

(Val 有)
$$\frac{\Gamma \vdash M : A_1 \times A_2 \quad \Gamma, x_1{:}A_1, x_2{:}A_2 \vdash N : B}{\Gamma \vdash (\text{有}\ (x_1{:}A_1, x_2{:}A_2)\ := M\ \text{做}\ N) : B}$$

联合类型（表格10）经常被忽视，但对于表达能力来说和乘积类型一样重要。 联合类型的元素 A_1+A_2可以被看作是带有 *left*标记的 A_1元素（由 *inLeft*创建），或者带有 *right*标记的 A_2元素（由*inRight*创建）。 可以通过 *isLeft*和 *isRight*来测试标记，并通过*asLeft*和 *asRight*提取相应的值。 如果错误地将 *asLeft*应用于带有右标记的值，将产生一个被捕获的错误或异常；这个被捕获的错误不被视为禁止错误。 请注意，可以安全地假设 *asLeft*的任何结果都具有类型 A_1，因为参数要么是左标记，结果确实是 A_1类型，要么是右标记，结果就不存在。 下标用于消除一些规则的歧义，就像我们在条件语句的情况下讨论的那样。规则（值为Case）描述了一种优雅的结构，可以替代 *isLeft*、 *isRight*、 *asLeft*、*asRight*以及相关的被捕获错误。 （它还消除了联合操作对布尔类型的任何依赖）。 根据 *M*的标签，case语句执行两个分支中的一个，将 *M*的未标记内容绑定到 N_1或 N_2的作用域中的 x_1或 x_2中。 垂直线将分支分隔开。

表格10. 联合类型

(类型 联合)
$$\frac{\Gamma \vdash A_1 \quad \Gamma \vdash A_2}{\Gamma \vdash A_1+A_2}$$

(Val 在左)
$$\frac{\Gamma \vdash M_1 : A_1 \quad \Gamma \vdash A_2}{\Gamma \vdash \text{在左}_{A_2}\ M_1 : A_1+A_2}$$

(Val 在右)
$$\frac{\Gamma \vdash A_1 \quad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \text{在右}_{A_1}\ M_2 : A_1+A_2}$$

(Val 是左)
$$\frac{\Gamma \vdash M : A_1+A_2}{\Gamma \vdash \text{是左}\ M : \text{布尔}}$$

(Val 是右)
$$\frac{\Gamma \vdash M : A_1+A_2}{\Gamma \vdash \text{是右}\ M : \text{布尔}}$$

(Val 作为左)
$$\frac{\Gamma \vdash M : A_1+A_2}{\Gamma \vdash \text{作为左}\ M : A_1}$$

(Val 作为右)
$$\frac{\Gamma \vdash M : A_1+A_2}{\Gamma \vdash \mathit{asRight}\ M : A_2}$$

(Val Case)
$$\frac{\Gamma \vdash M : A_1+A_2 \quad \Gamma, x_1{:}A_1 \vdash N_1 : B \quad \Gamma, x_2{:}A_2 \vdash N_2 : B}{\Gamma \vdash (\mathit{case}_B\ M\ \mathit{of}\ x_1{:}A_1\ \mathit{then}\ N_1 \mid x_2{:}A_2\ \mathit{then}\ N_2) : B}$$

在表达能力方面（如果不考虑实现），注意类型 *Bool*可以定义为 *Unit* + *Unit*，这样 *case*构造就简化为条件语句。类型 *Int*可以定义为 *Nat* + *Nat*，其中一个副本用于非负整数，另一个用于负数。 我们可以将一个典型的陷入错误定义为 $error_A = asRight(inLeft_A(unit)) : A$.
因此，我们可以为每种类型构建一个错误表达式。

产品类型和联合类型可以迭代生成元组类型和多个联合类型。
然而，这些派生类型非常不方便，很少在语言中见到。 相反，使用标记的产品和联合类型：它们分别被称为记录类型和变体类型。

记录类型是熟悉的命名类型集合，具有按名称提取组件的值级操作。 表11中的规则假设记录类型和记录的语法标识，可以重新排序它们的标记组件；这类似于函数的语法标识，可以重命名绑定变量。

在（Val Record With）中，产品类型的*with*语句被推广为记录类型。记录M的组件l_1，...，ln与作用域中的变量x_1，...，xn绑定。 Pascal也有类似的构造，也称为*with*，但绑定变量是隐式的。 （这导致作用域依赖于类型检查，从而导致由于隐藏的变量冲突而引起难以追踪的错误。）产品类型$A_1 \times A_2$可以定义为Record（first：A_1，second：A_2）。

表11. 记录类型

（类型记录） （l_i distinct）

$$\frac{\Gamma \vdash A_1 \quad ... \quad \Gamma \vdash A_n}{\Gamma \vdash \text{记录} \quad (l_1{:}A_1, ..., l_n{:}A_n)}$$

（值记录） （l_i distinct）

$$\frac{\Gamma \vdash M_1 : A_1 \quad ... \quad \Gamma \vdash M_n : A_n}{\Gamma \vdash \text{记录} \quad (l_1{=}M_1, ..., l_n{=}M_n) : \text{记录} \quad (l_1{:}A_1, ..., l_n{:}A_n)}$$

(值 记录 选择)

$$\frac{\Gamma \vdash M : \text{记录} \quad (l_1{:}A_1, ..., l_n{:}A_n) \qquad j \in 1..n}{\Gamma \vdash M.l_j : A_j}$$

(值 记录 带)

$$\frac{\Gamma \vdash M : \text{记录} \quad (l_1{:}A_1, ..., l_n{:}A_n) \qquad \Gamma, x_1{:}A_1, ..., x_n{:}A_n \vdash N : B}{\Gamma \vdash (\text{带} \ (l_1{=}x_1{:}A_1, ..., l_n{=}x_n{:}A_n){:=}M \ \text{做} \ N) : B}$$

变体类型（表12）是类型的命名不相交的联合；它们在语法上是可识别的，可以重新排列组件。 这个是l构造泛化了 *isLeft*和 *isRight*，而 *asl*构造泛化了 *asLeft*和 *asRight*。与联合一样，这些结构可以被一个 *case*语句替代，现在有多个分支。

联合类型 A_1+A_2可以定义为 *Variant*（*left*:A_1, *right*:A_2）。 枚举类型，例如 {*red*, *green*, *blue*}，可以定义为 *Variant*（*red*:*Unit*, *green*:*Unit*, *blue*:*Unit*）。

表格12. 变体类型

（类型 变体）（l_i distinct）

$$\frac{\Gamma \vdash A_1 \quad ... \quad \Gamma \vdash A_n}{\Gamma \vdash \text{变体} \quad (l_1{:}A_1,\ ...,\ l_n{:}A_n)}$$

（Val 变体）（l_i distinct）

$$\frac{\Gamma \vdash A_1 \quad ... \quad \Gamma \vdash A_n \qquad \Gamma \vdash M_{\mathrm{j}} : A_{\mathrm{j}} \qquad j \in 1..n}{\Gamma \vdash \text{变体}_{(1:A_1,\ ...,\mathrm{j}:A\mathrm{j})}(\ \mathrm{j}{=}M\ \mathrm{j}) : \text{变体}(_1{:}A_1,\ ...,\ \mathrm{j}{:}A\ \mathrm{j})}$$

（Val变体是）

$$\frac{\Gamma \vdash M : \text{变体}(_1{:}A_1,\ ...,\ \mathrm{j}{:}A\ \mathrm{j}) \qquad j \in 1..n}{\Gamma \vdash M \text{是} l_j : \text{布尔}}$$

（Val变体作为）

$$\frac{\Gamma \vdash M : \text{变体}(_1{:}A_1,\ ...,\ \mathrm{j}{:}A\ \mathrm{j}) \qquad j \in 1..n}{\Gamma \vdash M \text{作为} l_j : A_j}$$

（Val变体情况）

$$\frac{\Gamma \vdash M : \text{变体}(_1{:}A_1,\ ...,\ \mathrm{j}{:}A\ \mathrm{j}) \qquad \Gamma, x_1{:}A_1 \vdash N_1 : B \quad ... \quad \Gamma, x_n{:}A_n \vdash N_n : B}{\Gamma \vdash (\text{情况}_B M \text{的} l_1{=}x_1{:}A_1 \text{然后} N_1 \mid ...\ \mathrm{lj}{=}x\mathrm{j}{:}A\mathrm{j} \text{然后} N\mathrm{j}) : B}$$

引用类型（表格13）可以作为命令式语言中可变位置的基本类型。 Ref（A）的元素是一个包含类型为 A的可变单元。 （Val Ref）可以分配一个新的单元，（Val Assign）可以更新单元，（Val Deref）可以显式地取消引用。由于赋值的主要目的是执行副作用，所以其结果值被选择为 *unit*。

表格13. 引用类型

(类型引用)

$$\frac{\Gamma \vdash A}{\Gamma \vdash \text{引用} A}$$

(值引用)

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \text{引用}\ M : \text{引用}\ A}$$

(值解引用)

$$\frac{\Gamma \vdash M : \text{引用}\ A}{\Gamma \vdash \text{解引用}\ M : A}$$

(值赋值)

$$\frac{\Gamma \vdash M : \text{引用}\ A \qquad \Gamma \vdash N : A}{\Gamma \vdash M := N : Unit}$$

常见的可变类型可以从 *Ref*派生。 例如，可变记录类型可以建模为包含 *Ref*类型的记录类型。

表格14. 数组的实现

数组 (A)	数组类型
$Nat \times (Nat \to \text{引用}(A))$	一个边界加上一个从小于边界的索引到引用的映射

数组 $_A(N,M)$ 数组构造器 (用于 N个初始化为 M的引用)

let $cell_0 : Ref(A) = ref(M)$ *and ... and* $cell_{N-1} : Ref(A) = ref(M)$

in N, $\lambda x{:}Nat.$如果 x=0则 $cell_0$否则如果...否则如果 x=N-1则 $cell_{N-1}$否则错误$_{Ref(A)}$

bound(M) 数组边界

第一个M

$M[N]_A$ 数组索引

如果N < 第一个M

则解引用 ((第二个 M)(N))

否则报错$_A$

$M[N]$:= P 数组更新

如果N < 第一个M

则 ((第二个 M)(N)) := P

否则报错$_{Unit}$

更有趣的是，数组和数组操作可以建模如表14所示，其中 *Ar-ray*(A)是长度为 A类型元素的数组的类型。 (代码使用了一些算术原语和局部 *let*声明。) 当然，表14中的代码是一种低效的数组实现，但它说明了一个问题：更复杂构造的类型规则可以从更简单构造的类型规则推导出来。 表15中显示的数组操作的类型规则可以根据乘积、函数和引用的规则很容易地从表14中推导出来。

表15. 数组类型（派生规则）

(数组类型)

$$\frac{\Gamma \vdash A}{\Gamma \vdash \text{数组 } (A)}$$

(值 数组)

$$\frac{\Gamma \vdash N : \text{自然数} \quad \Gamma \vdash M : A}{\Gamma \vdash \text{数组 } (N,M) : \text{数组 } (A)}$$

(值 数组边界)

$$\frac{\Gamma \vdash M : \text{数组 } (A)}{\Gamma \vdash \text{边界} M : \text{自然数}}$$

(值 数组索引)

$$\frac{\Gamma \vdash N : \text{自然数} \quad \Gamma \vdash M : \text{数组 } (A)}{\Gamma \vdash M[N] : A}$$

(值 数组更新)

$$\frac{\Gamma \vdash N : \text{自然数} \quad \Gamma \vdash M : \text{数组 } (A) \quad \Gamma \vdash P : A}{\Gamma \vdash M[N] := P : Unit}$$

在大多数编程语言中，类型可以递归定义。 递归类型很重要，因为它们使得所有其他类型构造更有用。 它们通常是隐式引入的，或者没有明确的解释，并且它们的特性相当微妙。
因此，它们的形式化需要特别小心处理。

递归类型的处理需要对F_1进行相当基本的扩展：环境被扩展以包括类型变量 X。 这些类型变量用于递归类型的

A(表16)，直观地表示形式为$X=A$的递归方程的解其中 X可能出现在 A中。 操作展开和折叠是显式的强制转换，用于映射递归类型 $\mu X.A$及其展开 $[\mu X.A/X]A$（其中 $[B/X]A$是将 B替换为A中所有自由出现的X的替换），反之亦然。 这些强制转换在运行时没有任何效果（即展开(折叠(M))=M和折叠(展开(M'))=M'）。 它们通常在实际编程语言的语法中被省略，但它们的存在使得形式化处理更容易。

表16. 递归类型

(环境 X)

$$\frac{\Gamma \vdash \diamond \quad X \notin dom(\Gamma)}{\Gamma, X \vdash \diamond}$$

(类型记录)

$$\frac{\Gamma, X \vdash A}{\Gamma \vdash \mu X.A}$$

(值 折叠)

$$\frac{\Gamma \vdash M : [\mu X.A/X]A}{\Gamma \vdash \text{折叠}_{\mu X.A}\ M : \mu X.A}$$

(值 展开)

$$\frac{\Gamma \vdash M : \mu X.A}{\Gamma \vdash \text{展开}\ _{\mu X.A}\ M : [\mu X.A/X]A}$$

递归类型的一个常见应用是与乘积类型和并集类型一起定义列表和树的类型。 类型 $List_A$表示元素类型为 A的列表，在表17中定义了该类型，同时还定义了列表的构造函数 *nil*和 *cons*，以及列表分析器 *listCase*。

表17. 列表类型

列表_A $\quad \mu X.\text{单元}+(A\times X)$

空_A :列表$_A$ $\quad$ 折叠(在左单元)

$\text{连接}_A : A\to\text{列表}_A\to\text{列表}_A$ $\quad$ 折叠(在右头,尾)

$\text{列表情况}_{A,B}$:列表$_A\to B\to(A\times\text{列表}_A\to B)\to B$

$\lambda l{:}List_A.\ \lambda n{:}B.\ \lambda c{:}A\times List_A\to B.$

case (展开 l) *of unit*:*Unit then* $n \mid p{:}A\times List_A$ *then* $c\ p$

递归类型可以与记录类型和变体类型一起使用，以定义复杂的树结构，例如抽象语法树。 然后可以使用 *case*和 *with*语句方便地分析这些树。

与函数类型结合使用时，递归类型具有出人意料的表达能力。
通过巧妙的编码，可以证明值级别的递归已经隐含在递归类型中：无需引入递归作为单独的构造。 此外，在存在递归类型的情况下，可以在类型化语言中进行无类型编程。 更准确地说，表18展示了如何为任意类型 A定义一个发散元素 $_A$of和一个该类型的不动点运算符 $_A$。 表19展示了如何在类型化演算中编码无类型的 λ-演算。 （这些编码是针对按名调用的；在按值调用中，它们的形式稍有不同。）

表18. 通过递归类型编码发散和递归

$_A : A$	$(\lambda x{:}B.\ (\text{展开}_B\ x)\ x)\ (\text{折叠}_B\ (\lambda x{:}B.\ (\text{展开}_B\ x)\ x))$
$_A : (A\to A)\to A$	$X\to A$，对于任意的A

表19. 通过递归类型编码无类型 λ-演算

V	$\mu X.X\to X$	无类型 λ-项的类型
$\langle\!\langle x\rangle\!\rangle$	x	从无类型 λ-项到 V元素的翻译 $\langle\!\langle$-$\rangle\!\rangle$
$\langle\!\langle M\ N\rangle\!\rangle$	折叠$_V$ $(\lambda x{:}V.\langle\!\langle M\rangle\!\rangle)$ (展开 $_V\langle\!\langle M\rangle\!\rangle)\langle\!\langle N\rangle\!\rangle$	

在递归类型存在的情况下，类型等价特别有趣。 我们通过不处理类型定义，要求递归类型与其展开之间的显式折叠展开强制转换，以及不假设递归类型之间的任何标识（除了绑定变量的重命名）来回避了几个问题。 在当前的表述中，我们不需要为类型等价定义一个正式的判断：两个递归类型仅在结构上相同（除了绑定变量的重命名）时才等价。

这种简化的方法可以扩展到包括类型定义和类型等价性，直到递归类型的展开[2,26]。

用于命令式语言的一阶类型系统

命令式语言的类型系统有稍微不同的风格，主要是因为它们区分命令（不产生值）和表达式（产生值）。 （将命令转化为类型为*U*nit的表达式是完全可能的，但我们更倾向于保持自然的区分。）

作为命令式语言的类型系统的示例，我们考虑在表20中总结的无类型命令式语言。 这种语言允许我们研究声明的类型规则，这是我们迄今为止没有考虑过的。在这种语言中，过程和数据类型的处理非常简单，但是在第3节中描述的函数和数据的规则可以很容易地进行调整。 命令式语言的特性的含义应该是不言自明的。

表格20. 命令式语言的语法

A ::=	类型
布尔	布尔类型
自然数	自然数类型
过程	过程类型（无参数，无结果）

D ::=	声明
proc $I = C$	过程声明
var $I : A = E$	变量声明
C ::=	命令
$I := E$	赋值
$C_1; C_2$	顺序组合
begin D in C end	块
call I	过程调用
while E do C end	while循环
E ::=	表达式
I	标识符
N	数字
$E_1 + E_2$	两个数字的和
E_1 not= E_2	两个数字的不等式

我们命令式语言的判断列在表21中。判断 $\Gamma \vdash C$和 $\Gamma \vdash E : A$对应于单一判断 $\Gamma \vdash M : AF_1$，因为我们现在区分了命令 C和表达式 E。 判断 $\Gamma \vdash D \therefore S$将一个签名$S$分配给一个声明$D$；签名实际上是一个声明的类型。 在这个简单的语言中，一个签名由一个组件组成，例如 x :自然数，而一个匹配的声明可以是变量 x :自然数= 3。 一般来说，签名将由这些组件的列表组成，并且看起来非常相似或与环境 Γ相同。

表格21. 命令式语言的判断

$\Gamma \vdash \diamond$	Γ是一个良好形成的环境
$\Gamma \vdash A$	A是 Γ中的一个良好类型
$\Gamma \vdash C$	C是 Γ中的一个良好命令
$\Gamma \vdash E : A$	E是 Γ中类型为 A的一个良好表达式
$\Gamma \vdash D \therefore S$	D是 Γ中签名为 S的一个良好声明

表格22列出了命令式语言的类型规则。

表格22. 命令式语言的类型规则

(环境 ø)

$$\frac{}{\emptyset \vdash \diamond}$$

(环境我)

$$\frac{\Gamma \vdash A \quad 我 \notin 领域(\Gamma)}{\Gamma, 我 : A \vdash \diamond}$$

(布尔类型)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash 布尔}$$

(自然数类型)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash 自然数}$$

(类型 过程)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash 过程}$$

(声明过程)
$$\frac{\Gamma \vdash C}{\Gamma \vdash (\text{proc } I = C) \therefore (I : Proc)}$$

(声明变量)
$$\frac{\Gamma \vdash E : A \qquad A \in \{Bool, Nat\}}{\Gamma \vdash (\text{var } I : A = E) \therefore (I : A)}$$

(通信赋值)
$$\frac{\Gamma \vdash I : A \qquad \Gamma \vdash E : A}{\Gamma \vdash I := E}$$

(通信序列)
$$\frac{\Gamma \vdash C_1 \qquad \Gamma \vdash C_2}{\Gamma \vdash C_1 ; C_2}$$

(通信块)
$$\frac{\Gamma \vdash D \therefore (I : A) \qquad \Gamma, I{:}A \vdash C}{\Gamma \vdash \text{开始 } D \text{在 } C \text{结束}}$$

(通信调用)
$$\frac{\Gamma \vdash I : Proc}{\Gamma \vdash \text{调用 } I}$$

(通信循环)
$$\frac{\Gamma \vdash E : Bool \qquad \Gamma \vdash C}{\Gamma \vdash \text{当 } E \text{执行 } C \text{结束}}$$

(表达式 标识符)
$$\frac{\Gamma_1, I{:}A, \Gamma_2 \vdash \diamond}{\Gamma_1, I{:}A, \Gamma_2 \vdash I : A}$$

(表达式 数字)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash N : \text{自然数}}$$

(表达式 加)
$$\frac{\Gamma \vdash E_1 : \text{自然数} \quad \Gamma \vdash E_2 : \text{自然数}}{\Gamma \vdash E_1 + E_2 : \text{自然数}}$$

(表达式 不等于)
$$\frac{\Gamma \vdash E_1 : \text{自然数} \quad \Gamma \vdash E_2 : \text{自然数}}{\Gamma \vdash E_1 \text{不等于 } E_2 : \text{布尔值}}$$

规则 (环境 ...), (类型 ...), 和 (表达式 ...) 是我们在F_1中看到的规则的直接变体。 规则 (声明 ...) 处理声明的类型。 规则 (命令 ...) 处理命令; 注意 (命令 块) 如何将签名转换为环境的一部分，在检查块的主体时。

第五章 二阶类型系统

许多现代语言包括类型参数、类型抽象或两者的构造。 类型参数可以在几种语言的模块系统中找到，其中泛型模块、类或接口由稍后提供的类型参数化。 计划中的Java和C#扩展在类和接口级别使用类型参数。 (C++模板类似于类型参数，但实际上是一种宏展开形式，具有非常不同的属性。) 多态语言如ML和Haskell在函数级别更广泛地使用类型参数。 类型抽象可以与模块结合使用，在接口中以不透明类型的形式出现，例如Modula-2和Modula-3。CLU等语言在数据级别使用类型抽象，以获得抽象数据类型。 这些高级特性可以通过所谓的二阶类型系统进行建模。

二阶类型系统通过类型参数的概念扩展了一阶类型系统。 一种新的术语，写作 $\lambda X.M$,表示一个以类型变量 X为任意类型的参数化程序 M。 例如，对于一个固定类型 A的恒等函数，写作 $\lambda x{:}A.x$，可以通过抽象化将其转化为一个参数化的恒等函数

over $A$and writingid　$\lambda X.\lambda x{:}X.x$. 然后可以通过类型实例化 $id\,A$将这样的参数化函数实例化为任何给定的类型 A，该类型实例化写作id A，它返回 $\lambda x{:}A.x$.对应于新的项 $\lambda X.$

M我们需要新的全称量化类型。 像 $\lambda X.M$这样的项的类型写作 $\forall X.A$，表示对于所有的 X，体M具有类型A（这里 M和 A可能包含X的出现）。 例如，参数化身份的类型是 $id:\forall X.X\to X$，因为对于所有的 X，类型实例化$id\,X$具有类型 $X\to X$。

纯二阶系统F_2（表23）仅基于类型变量、函数类型和量化类型。 请注意，我们正在放弃基本类型 K，因为我们现在可以使用类型变量作为基本情况。 事实证明，几乎任何感兴趣的基本类型都可以在F_2[4]中进行编码。 同样，乘积类型、和类型、存在类型和一些递归类型可以在F_2中进行编码：多态具有惊人的表达能力。 因此，从技术上讲，没有必要直接处理这些类型构造。

表23. F_2的语法

A,B ::=	类型
X	类型变量
$A\to B$	函数类型
$\forall X.A$	全称量化类型
M,N ::=	项
x	变量
$\lambda x{:}A.M$	函数
$M\ N$	应用
$\lambda X.M$	多态抽象
$M\ A$	类型实例化

F_2类型和项的自由变量可以按照通常的方式定义；简而言之， $\forall X$。F_2的一个有趣方面是在类型实例化的类型规则（Val Appl2）中替换类型变量的类型。

表格24. F_2的判断

$\Gamma\vdash\ \diamond$	Γ是一个良好形成的环境
$\Gamma\vdash A$	A是在 Γ中形成良好的类型
$\Gamma\vdash M:A$	M是一个类型为 A的良好形式术语在 Γ

表格25. F_2的规则

(环境 $\emptyset$)　(环境 x)　(环境 X)

$$\frac{}{\emptyset\vdash\ \diamond}\qquad \frac{\Gamma\vdash A\quad {}_{x\notin\text{属于}(\Gamma)\text{的定义域}}}{\Gamma,x{:}A\vdash\ \diamond}\qquad \frac{\Gamma\vdash\ \diamond\quad X\notin dom(\Gamma)}{\Gamma,X\vdash\ \diamond}$$

(类型 X)
$$\frac{\Gamma', X, \Gamma'' \vdash \diamond}{\Gamma', X, \Gamma'' \vdash X}$$

(类型箭头)
$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \to B}$$

(类型 Forall)
$$\frac{\Gamma, X \vdash A}{\Gamma \vdash \forall X.A}$$

(值 x)
$$\frac{\Gamma', x{:}A, \Gamma'' \vdash \diamond}{\Gamma', x{:}A, \Gamma'' \vdash x{:}A}$$

(值函数)
$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : A \to B}$$

(值应用)
$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

(值 Fun2)
$$\frac{\Gamma, X \vdash M : A}{\Gamma \vdash \lambda X.M : \forall X.A}$$

(值 Appl2)
$$\frac{\Gamma \vdash M : \forall X.A \quad \Gamma \vdash B}{\Gamma \vdash M\ B : [B/X]A}$$

F_2的判断(Table $_{24)}$)与F_1的判断相同，但环境更丰富。相对于F_1，新规则(Table $_{25)}$)如下：(Env X)，将类型变量添加到环境中；(Type Forall)，构造一个量化类型 $\forall X.$从类型变量 X和类型A构建一个 A(Afrom a type variable X anda type$A$ where$X$may occur)；(Val Fun2)，构建一个多态抽象；(ValAppl2)，将多态抽象实例化为给定类型，其中[B/ X] A是将 B替换为 A中所有自由出现的X的替换。 例如，如果 id具有类型 $\forall X.X \to X$和A是一个类型，那么根据(Val Appl2)，我们有 idA具有类型[A/X]($X \to X$) ≡$A \to$A。 作为一个简单但有教育意义的练习，读者可以构建 $id(\forall X.X \to X)(id)$的推导。作为$F_2$的扩展，我们可以采用所有我们已经讨论过的一阶构造。 一个更有趣的扩展是存在量化类型，也

称为类型抽象：

表格 26. 存在类型

(类型存在)
$$\frac{\Gamma, X \vdash A}{\Gamma \vdash \exists X.A}$$

(值打包)
$$\frac{\Gamma \vdash [B/X]M : [B/X]A}{\Gamma \vdash (pack_{\exists X.A}\ X{=}B\ with\ M) : \exists X.A}$$

(值打开)
$$\frac{\Gamma \vdash M : \exists X.A \quad \Gamma, X, x{:}A \vdash N : B \quad \Gamma \vdash B}{\Gamma \vdash (open_B\ M\ as\ X, x{:}A\ in\ N) : B}$$

为了说明存在类型的使用，我们考虑一个用于布尔值的抽象类型。 正如我们之前所说的，布尔值可以表示为类型 Unit+ Unit。 现在我们可以展示如何隐藏这个表示细节，使得不关心布尔值如何实现的客户端可以使用 true、false 和 cond（条件语句）。 我们首先为这样一个客户端定义一个接口，

$Bool$接口　　$\exists Bool.$记录($true$: $Bool$, $false$: $Bool$, $cond$: $\forall Y.\ Bool \to Y \to Y \to Y$)

这个接口声明存在一种类型 *Bool*（不透露其身份），支持适当类型的操作 *true*, *false*和cond。 条件语句的结果类型 *Y*是参数化的，可能根据使用的上下文而变化。

接下来，我们定义了这个接口的一个特定实现；它将 *Bool*表示为*Unit+Unit*，并通过一个case语句实现条件语句。 布尔表示类型和相关的布尔操作通过 *pack*结构打包在一起。

$$
\begin{array}{l}
boolModule : BoolInterface \\
\quad pack_{BoolInterface}\ Bool{=}Unit{+}Unit \\
\quad with\ \ record(\\
\qquad\qquad true = inLeft(unit), \\
\qquad\qquad false = inRight(unit), \\
\qquad\qquad cond = \lambda Y.\ \lambda x{:}Bool.\ \lambda y_1{:}Y.\ \lambda y_2{:}Y. \\
\qquad\qquad\qquad\qquad case_Y\ x\ of\ x_1{:}Unit\ then\ y_1 \mid x_2{:}Unit\ then\ y_2)
\end{array}
$$

最后，客户端可以通过打开它来使用这个模块，从而获得对布尔类型的抽象名称 *Bool*和布尔操作记录的名称 *boolOp*的访问权限。 这些名称在下一个示例中用于返回一个自然数的简单计算。 （后面的计算 *in*实质上是*if boolOp.true then* 1 *else* 0。）

$$
\begin{array}{l}
open_{Nat}\ boolModule \\
作为 Bool,\ boolOp{:}记录(true{:}\ Bool,\ false{:}\ Bool,\ cond{:}\ \forall Y.\ Bool{\to}Y{\to}Y{\to}Y) \\
in\ boolOp.cond(Nat)(boolOp.true)(1)(0)
\end{array}
$$

读者应该根据之前给出的规则验证这些示例的类型检查。
请注意(Val Open)的关键第三个假设，它意味着结果类型 *B*不能包含变量 *X*。 这个假设禁止例如将 *boolOp*.作为open的主体（在前面的例子中），因为结果类型将会是变量*Bool*。由于第三个假设，表示类型的抽象名称（*Bool*）不能逃离open的作用域，因此具有表示类型的值也不能逃离。 这种限制是必要的，否则表示类型可能会被客户端知晓。
这种限制是必要的，否则表示类型可能会被客户端知晓。

6 子类型

带类型的面向对象语言具有特别有趣和复杂的类型系统。 关于这些语言的特征几乎没有共识，但至少有一个特征几乎普遍存在：子类型。 子类型捕捉了类型之间包含关系的直观概念，其中类型被视为值的集合。 类型的元素也可以被视为其任何超类型的元素，从而允许一个值（对象）在许多不同的类型上下文中灵活使用。

在考虑子类型关系时，比如在面向对象的编程语言中，习惯上会添加一个新的判断 $\Gamma \vdash A <: B$，表示 *A*是*B*的子类型（表27）。 直觉上， *A*的任何元素都是 *B*的元素，或者更恰当地说，任何类型为 *A*的程序也是类型为 *B*的程序。

最简单的子类型化类型系统之一是F_1的扩展，称为$F_{1<:}$。 F_1的语法与以前相同，只是添加了一个类型 *Top*，它是所有类型的超类型。 现有的类型规则也没有改变。 子类型判断是独立公理化的，并且添加了一个称为**subsumption**的单一类型规则，将类型判断与子类型判断连接起来。

表27. 具有子类型的类型系统的判断

$\Gamma \vdash \diamond$	Γ是一个良好形成的环境
$\Gamma \vdash A$	A是在 Γ中形成良好的类型
$\Gamma \vdash A <: B$	在 Γ中，A是B的子类型
$\Gamma \vdash M : A$	M是一个类型为 A的良好形式术语在 Γ

子类型规则指出，如果一个术语具有类型 A，并且 A是 B的子类型，则该术语也具有类型 B。 也就是说，当类型成员资格被视为集合成员资格时，子类型行为非常类似于集合包含关系。

表28中的子类型关系被定义为一个自反且传递的关系，其中有一个被称为 *Top*的最大元素，因此被解释为所有良好类型术语的类型。

函数类型的子类型关系表明，如果 A'是 A的子类型，并且B是B'的子类型，则 $A \to B$是 $A' \to B'$的子类型。 请注意，对于函数参数，包含关系是反转的（逆变的），而对于函数结果，它是相同方向的（协变的）。 简单的思考揭示了这是唯一合理的规则。 类型为 $A \to B$的函数 M接受类型为 A的元素；显然它也接受任何 A的子类型 A'的元素。同样的函数
M返回类型为 B的元素；显然，它返回属于任何超类型 B' of$B$的元素。 因此，任何类型为 $A \to B$的函数，通过接受类型为 A'的参数和返回类型为 B'的结果，也具有类型 $A' \to B'$。 后者与 $A \to B$是 $A' \to B'$的子类型是兼容的。

通常，我们说类型变量在F_1的另一个类型中逆变出现，如果它总是出现在奇数个箭头的左边（双逆变等于协变）。 例如， $X \to Unit$和（$Unit \to X$）$\to Unit$在 X中是逆变的，而 $Unit \to X$和（$X \to Unit$）$\to X$在 X中是协变的。

表格28. $F_{1<:}$的附加规则

(Sub Refl)
$$\frac{\Gamma \vdash A}{\Gamma \vdash A <: A}$$

(Sub Trans)
$$\frac{\text{在}\Gamma\text{中，}\vdash A <: B \quad \text{在}\Gamma\text{中，}\vdash B <: C}{\text{在}\Gamma\text{中，}\vdash A <: C}$$

(Val Subsumption)
$$\frac{\text{在}\Gamma\text{中，}\vdash a : A \quad \text{在}\Gamma\text{中，}\vdash A <: B}{\text{在}\Gamma\text{中，}\vdash a : B}$$

(Type Top)
$$\frac{\Gamma \vdash \diamond}{\text{在}\Gamma\text{中，}\vdash Top}$$

(Sub Top)
$$\frac{\Gamma \vdash A}{\text{在}\Gamma\text{中，}\vdash A <: Top}$$

(Sub Arrow)
$$\frac{\text{在}\Gamma\text{中，}\vdash A' <: A \quad \text{在}\Gamma\text{中，}\vdash B <: B'}{\text{在}\Gamma\text{中，}\vdash A \to B <: A' \to B'}$$

可以在基本类型上添加特定的子类型规则，例如 *Nat* <: *Int* [19]。

我们考虑的所有结构化类型都可以作为F_1的扩展，都有简单的子类型规则；因此，这些结构化类型也可以添加到$F_{1<:}$中（表格29）。 通常，我们需要为每个类型构造器添加一个子类型规则，确保该子类型规则与子汇总一起是正确的。 对于乘积和并集，子类型规则逐个组件进行操作。 对于记录和变体，子类型规则也按长度进行操作：较长的记录类型是较短的记录类型的子类型（通过子类型可以忽略附加字段），而较短的变体类型是较长的变体类型的子类型（通过子类型可以引入附加案例）。例如，

工作年龄　　变体(学生:单位, 成年人:单位)
年龄　　变体(儿童:单位, 学生:单位, 成年人:单位, 老年人:单位)
工人　　记录(姓名:字符串,年龄:工作年龄,职业:字符串)
人　　记录(姓名:字符串,年龄:年龄)

然后,

工作年龄 <:年龄
工人　<: 人

引用类型没有任何子类型规则：Ref(A) <：Ref(B)仅在$A = B$的情况下成立（在这种情况下，Ref(A) <：Ref(B)是由于自反性）。 这个严格的规则是必要的，因为引用既可以读取又可以写入，因此既可以协变又可以逆变。
出于同样的原因，数组类型没有额外的子类型规则。

表29. F的扩展的附加规则$_{1<:}$

（子产品）

$$\frac{\Gamma \vdash A_1 <: B_1 \quad \Gamma \vdash A_2 <: B_2}{\Gamma \vdash A_1 \times A_2 <: B_1 \times B_2}$$

（子联合）

$$\frac{\Gamma \vdash A_1 <: B_1 \quad \Gamma \vdash A_2 <: B_2}{\Gamma \vdash A_1 + A_2 <: B_1 + B_2}$$

（子记录）（l_i distinct)

$$\frac{\Gamma \vdash A_1 <: B_1 \quad \ldots \quad \Gamma \vdash A_n <: B_n \quad \Gamma \vdash A_{n+1} \quad \ldots \quad \Gamma \vdash A_{n+m}}{\Gamma \vdash \text{记录} \quad (l_1{:}A_1, \ldots, l_{n+m}{:}A_{n+m}) <: \text{记录} \quad (l_1{:}B_1, \ldots, l_n{:}B_n)}$$

(子变体) (l_i 不同)

$$\frac{\Gamma \vdash A_1 <: B_1 \quad \ldots \quad \Gamma \vdash A_n <: B_n \quad \Gamma \vdash B_{n+1} \quad \ldots \quad \Gamma \vdash B_{n+m}}{\Gamma \vdash \text{变体} \quad (l_1{:}A_1, \ldots, l_n{:}A_n) <: \text{变体} \quad (l_1{:}B_1, \ldots, l_{n+m}{:}B_{n+m})}$$

与F1的情况一样，考虑递归类型时需要对环境的结构进行更改。 这次，我们必须向环境中添加有界变量（表30）。
由*Top*绑定的变量对应于我们旧的无约束变量。 递归类型的子类型规则（Sub Rec）的正确性（表31）并不明显，但直观上是相当直接的。为了检查$\mu X.A <: \mu Y.B$，我们假设$X <: Y$，并检查$A <: B$；这个假设在查找A和B中的匹配出现时对我们有帮助，只要它们在协变的上下文中。一个更简单的规则断言，对于任何X，$\mu X.A <: \mu X.B$只要$A <: B$即可，但是 B whenever$A$ <:$B$for any$X$, but

当 X出现在逆变上下文中时，此规则是不可靠的（例如，紧接在箭头的左边）。

表格30. 具有有界变量的环境

(环境 $X<:$)	(类型 $X<:$)	(子类型 $X<:$)
$\dfrac{\Gamma \vdash A \quad X \notin dom(\Gamma)}{\Gamma, X<:A \vdash \diamond}$	$\dfrac{\Gamma', X<:A, \Gamma'' \vdash \diamond}{\Gamma', X<:A, \Gamma'' \vdash X}$	$\dfrac{\Gamma', X<:A, \Gamma'' \vdash \diamond}{\Gamma', X<:A, \Gamma'' \vdash X<:A}$

表格31. 子类型递归类型

(类型记录)	(子类型 递归)
$\dfrac{\Gamma, X<:\text{顶部} \vdash A}{\Gamma \vdash \mu X.A}$	$\dfrac{\Gamma \vdash \mu X.A \quad \Gamma \vdash \mu Y.B \quad \Gamma, Y<:\text{顶部}, X<:Y \vdash A <: B}{\Gamma \vdash \mu X.A <: \mu Y.B}$

环境中的有界变量也是F_2扩展的基础，该扩展提供了一个称为$F_{2<:}$的系统（表格32）。在此系统中，术语 $\lambda X<:A.M$表示一个相对于类型变量 X的任意子类型的程序M。这是F_2的一般化，因为F_2术语 $\lambda X.M$可以表示为 $\lambda X<:Top.M$。对应于术语 $\lambda X<: A.M$，我们有形式为 $\forall X<:A.B$的有界类型量词。

表32. $F_{2<:}$的语法

A,B ::=	类型
X	类型变量
顶部	最大类型
$A \to B$	函数类型
$\forall X<:A.B$	有界全称量化类型
M,N ::=	项
x	变量
$\lambda x:A.M$	函数
$M\ N$	应用
$\lambda X<:A.M$	有界多态抽象
$M\ A$	类型实例化

$F_{2<:}$的作用域和项的定义与F_2类似，只是在A中没有绑定X。B在B中绑定X，但在A中没有绑定X。M在M中绑定X，但在A中没有绑定X。

$F_{2<:}$的类型规则包括$F_{1<:}$的大部分类型规则（即(Env $\emptyset$)，(Env x)，(Type Top)，(Type Arrow)，(Sub Refl)，(Sub Trans)，(Sub Top)，(Sub Arrow)，(Val Subsump-tion)，(Val x)，(Val Fun)和(Val Appl)），加上有界变量的规则（即(Env$X<:$)，(Type $X<:$)和(Sub $X<:$)），以及表33中列出的有界多态规则。

表33. 有界全称量词的规则

(类型Forall<:)
$$\frac{\Gamma, X<:A \vdash B}{\Gamma \vdash \forall X<:A.B}$$

(子类型Forall<:)
$$\frac{\text{在}\Gamma\text{中，} \vdash A' <: A \quad \Gamma, X<:A' \vdash B <: B'}{\Gamma \vdash (\forall X<:A.B) <: (\forall X<:A'.B')}$$

(Val Fun2<:)
$$\frac{\Gamma, X<:A \vdash M : B}{\Gamma \vdash \lambda X<:A.M : \forall X<:A.B}$$

(Val Appl2<:)
$$\frac{\Gamma \vdash M : \forall X<:A.B \quad \text{在}\Gamma\text{中，} \vdash A' <: A}{\Gamma \vdash M\ A' : [A'/X]B}$$

至于F_2，我们不需要在$F_{2<:}$中添加其他类型构造，因为所有常见的类型构造都可以在其中表示（除了递归）。 此外，事实证明，用于F_2的编码符合预期的子类型规则。例如，可以编码有界存在类型，以满足表34中描述的规则。类型 $\exists X<:A.B$表示部分抽象类型，其表示类型 X不完全已知，但已知是 A的子类型。 这种部分抽象在一些基于子类型的语言中出现（例如Modula-3）。

表34. 有界存在量词的规则（可推导）

(Type Exists<:)
$$\frac{\Gamma, X<:A \vdash B}{\Gamma \vdash \exists X<:A.B}$$

(Sub Exists<:)
$$\frac{\Gamma \vdash A <: A' \quad \Gamma, X<:A \vdash B <: B'}{\Gamma \vdash (\exists X<:A.B) <: (\exists X<:A'.B')}$$

(Val Pack<:)
$$\frac{\Gamma \vdash C <: A \quad \Gamma \vdash [C/X]M : [C/X]B}{\Gamma \vdash (pack_{\exists X<:A.B}\ X<:A=C\ with\ M) : \exists X<:A.B}$$

(Val Open<:)
$$\frac{\Gamma \vdash M : \exists X<:A.B \quad \Gamma \vdash D \quad \Gamma, X<:A, x:B \vdash N : D}{\Gamma \vdash (open_D\ M\ as\ X<:A, x:B\ in\ N) : D}$$

在$F_{2<:}$中，需要进行一些非平凡的工作来获得记录和变体类型的编码，以满足预期的子类型规则，但即使这些规则也可以找到[6]。

7 等价性

为了简单起见，我们避免描述某些在类型系统变得复杂时必要的判断，以及在捕捉程序语义以及它们的类型时所需的判断。 我们简要讨论了其中一些判断。

类型等价判断的形式为 $\Gamma \vdash A = B$，当类型等价性是非平凡的且需要精确描述时可以使用。 例如，某些类型系统将递归类型及其展开进行了标识，此时我们将有 $\Gamma \vdash \mu X.\ A = [\mu X.A/X]A$，只要 $\Gamma \vdash \mu X.A$。

作为另一个例子, 具有类型运算符 $\lambda X.$的类型系统A(从类型到类型的函数) 有一个形如 $\Gamma \vdash (\lambda X.A)\ B = [A/X]B$的规约规则. 类型等价判断通常在重新类型化规则中使用, 规定如果 $\Gamma \vdash M : A$且 $\Gamma \vdash A = B$则 $\Gamma \vdash M : B$.

一个术语等价判断决定了哪些程序在公共类型下是等价的。 它的形式为 $\Gamma \vdash M = N : A$。 例如，通过适当的规则，我们可以确定 $\Gamma \vdash 2+1 = 3 : Int$。 术语等价判断可以用于为程序提供有类型的语义：如果 N是一个不可约表达式，那么我们可以将 N视为程序 M的结果值。

8 类型推断

类型推断是在给定类型系统中为一个术语找到类型的问题，如果存在任何类型的话。 在我们之前考虑的类型系统中，程序有丰富的类型注解。 因此，类型推断问题通常只是检查注解的相互一致性。 这个问题并不总是简单的，但是，就像F_1的情况一样，可能存在简单的类型检查算法。

一个更难的问题，称为可类型化或类型重构，是指从一个未标注类型的程序 M开始，找到一个环境 Γ，一个带有类型注释的版本 M'，以及一个类型
使得A成为 M'相对于 Γ的类型。 （一个带有类型注释的程序 M'是指将所有类型注释去除后还原为 M的程序。）在F_1中，通过ML中使用的Hindley-Milner算法可以解决未标注 λ-演算的类型重构问题[17]；此外，该算法具有以唯一方式表示 λ-项的所有可能的F_1类型的特性。 然而，对于未标注 λ-演算的类型重构问题，在F_2中是无法解决的[32]。 在具有子类型的系统中进行类型重构仍然是一个尚未解决的问题，尽管一些特殊的解决方案开始出现[1, 10, 13, 24]。

我们在这里集中讨论一些代表性系统的类型推断算法：F_1，F_2和 $F_{2<:}$。 前两个系统具有唯一类型属性：如果一个术语有一个类型，那么它只有一个类型。 在 $F_{2<:}$中没有唯一类型，仅仅因为包含关系规则将一个类型的所有超类型分配给具有该类型的任何术语。 然而，最小类型属性成立：如果一个术语有一组类型，那么该集合在子类型顺序中有一个最小元素[8]。
最小类型属性适用于许多常见的 $F_{2<:}$和 $F_{1<:}$的扩展，但在基本类型上存在特定子类型时可能失败。

类型推断问题

在给定的类型系统中，给定一个环境 Γ和一个术语 M，是否存在一个类型 A，使得 $\Gamma \vdash M : A$是有效的？ 以下是一些例子：

- 在F_1中，给定 $M \equiv \lambda x{:}K.$对于任意的x和任何良好形式的 Γ，我们有 $\Gamma \vdash M : K \to K$.
- 在F_1中，给定 $M \equiv \lambda x{:}K.$对于任意的$y(x)$ 和 $\Gamma \equiv \Gamma', y{:}K \to K$，我们有 $\Gamma \vdash M : K \to K$.
- 在F_1中，没有 $\lambda x{:}B$的类型。对于任何类型 B，$x(x)$ 都是合法的。
- 然而，在$F_{1<:}$中，有类型 $\Gamma \vdash \lambda x{:}Top \to B.$对于任何类型$B$，$x(x) : (Top \to B) \to B$，因为$x$也可以被赋予类型 Top.

- 此外，在具有递归类型的F_1中，存在类型 $\Gamma \vdash \lambda x{:}B.(展开_B\ x)(x) : B \to B$，
  对于 $B \equiv \mu X.X \to X$，因为展开$_B$ x的类型为 $B \to B$。
- 最后，在F_2中存在类型 $\Gamma \vdash \lambda x{:}B.X \to X$，因为 $x(B)$
  的类型为 $B \to B$。

（类型推断问题的另一种表述要求找到 Γ，而不是给定的g-en。然而，在编程实践中，人们只对嵌入在完整编程上下文中的程序的类型推断感兴趣，因此 Γ是已知的。）

我们从纯F_1的类型推断算法开始，表35中给出。 该算法可以直接扩展到之前研究的所有一阶类型结构。
这是Pascal和所有类似过程语言中使用的类型检查算法的基础。

主要程序 *Type*(Γ, *M*)接受一个环境 Γ和一个项 *M*，并产生 *M*的唯一类型（如果有）。指令 *fail*会导致算法全局失败：它表示类型错误。 在这个算法中，和后续的算法一样，我们假设初始环境参数 Γ是良好形成的，以排除将无效环境传递给内部调用的可能性。 （例如，当检查完整程序时，我们可以从空环境开始。）无论如何，我们可以很容易地编写一个子程序来检查环境的良好形成性，从我们提供的代码中。对于 $\lambda x{:}A.M$ 的情况应该有一个限制，要求 $x \notin dom(\Gamma)$，因为 x用于扩展 Γ。 然而，通过重命名，例如在运行算法之前使所有绑定器唯一，可以轻松地规避这种限制。 我们在表35、36和37中省略了这种限制。

表35. F_1的类型推断算法

类型(Γ, x)
　　如果$x{:}A \in \Gamma$对于某个A，则返回A，否则返回失败
类型(Γ, $\lambda x{:}A.M$)
　　$A \to Type((\Gamma, x{:}A), M)$
类型(Γ, $M\ \ N$)
　　如果 $Type(\Gamma, M) \equiv Type(\Gamma, N) \to B$ 对于某个 B，则返回 B，否则返回失败

举个例子，让我们考虑一下$\lambda z{:}K.$的类型推断问题在环境中，$y(z)$的类型是$y{:}K \to K$，我们在第3节中给出了一个完整的F_1推导 算法的步骤如下:

$Type((\emptyset, y{:}K \to K), \lambda z{:}K.y(z))$
$= K \to Type((\emptyset, y{:}K \to K, z{:}K), y(z))$
$= K \to (if\ Type((\emptyset, y{:}K \to K, z{:}K), y) \equiv Type((\emptyset, y{:}K \to K, z{:}K), z) \to B$ 对于某个
　　B，则返回 B，否则返回失败)
$= K \to (if\ K \to K \equiv K \to B$ 对于某个 B，则返回 B，否则返回失败）　　(taking $B \equiv K$)
$= K \to K$

F_2的类型推断算法(Table 36)比F_1的算法稍微复杂一些，但是它需要一个子程序*G*ood(Γ, *A*)来验证源程序中遇到的类型

是良好形式的。 这个检查是必要的，因为F_2中的类型可能包含未绑定的类型变量。 在类型实例化情况下，还必须使用替换子程序，$M\,A$。

表36。 F_2的类型推断算法

好(Γ, X) $X \in dom(\Gamma)$
好($\Gamma, A \to B$) 好 (Γ, A) 和好 (Γ, B)
好($\Gamma, \forall X.\, A$) 好 $((\Gamma, X), A)$

类型(Γ, x)
如果 $x{:}A \in \Gamma$对于某个A，则返回A，否则返回失败
类型($\Gamma, \lambda x{:}A.M$)
如果好(Γ, A)则 $A \to$类型($(\Gamma, x{:}A), M$)否则返回失败
类型($\Gamma, M\,N$)
如果类型(Γ, M) ≡类型(Γ, N)$\to B$对于某个B，则返回B，否则返回失败
类型($\Gamma, \lambda X.M$)
$\forall X.$类型($(\Gamma, X), M$)
类型($\Gamma, M\,A$)
对于某些X、B和$Good(\Gamma, A)$，如果 $[A/X]B$则成功，否则失败

给定在表37中的$F_{2<:}$的类型推断算法更加复杂。子程序 *Subtype*(Γ,*A, B*)试图确定 *A*是否是 *B*在 Γ中的子类型，并且乍一看是直接的。然而，已经证明 *Subtype*只是一个半算法：它可能在某些不是子类型关系的 *A*,*B*上发散。也就是说，$F_{2<:}$的类型检查器可能在类型不正确的程序上发散，尽管它仍然会收敛并为类型正确的程序产生最小类型。更一般地说，子类型关系没有决策过程：$F_{2<:}$的类型系统是不可判定的[25]。已经尝试过将$F_{2<:}$缩减为可判定的子集；目前最简单的解决方案是要求(Sub Forall<:)中的量词边界相等。无论如何，坏的 *A*,*B*对在实践中极不可能出现。该算法在通常意义上仍然是正确的：如果它找到一个类型，程序就不会出错。唯一棘手的情况是量词的子类型关系；算法对$F_{1<:}$的限制是可判定的并产生最小类型。

表格37. $F_{2<:}$的类型推断算法

好(Γ, X) $X \in dom(\Gamma)$
好($\Gamma,$ 顶部) 真
好($\Gamma, A \to B$) 好 (Γ, A) 和好 (Γ, B)
好($\Gamma, \forall X{<:}A.\, B$) 好 (Γ, A) 和好 $((\Gamma, X{<:}A), B)$

子类型(Γ, A,顶级)　　*true*
子类型(Γ, X, X)　　*true*
子类型(Γ, X, A)　　对于$A \neq X$,顶级
　　，如果$X<:B \in \Gamma$ 对于某个B，如果成立则子类型(Γ, B, A)否则为 *false*
子类型(Γ, $A \to B$, $A' \to B'$)
　　子类型(Γ, A', A) 和子类型　(Γ, B, B')
子类型(Γ B, $\forall X'<:A'.B'$)
　　子类型(Γ, A', A) 和子类型　((Γ, $X'<:A'$), $[X'/X]B$, B')
子类型(Γ, A, B)　　假　　否则

暴露(　Γ, X)　　如果 $X<:A \in \Gamma$对于某个 A，则暴露(Γ, A)否则失败
暴露(　Γ, A)　　A　　否则

类型(Γ, x)
　　如果 $x:A \in \Gamma$ 对于某个 A，则 A 否则失败
类型(Γ, $\lambda x:A.M$)
　　如果 *Good*(Γ, A)然后 $A \to$类型((Γ, x:A), M)否则失败
类型(Γ, M N)
　　如果暴露(Γ,类型(Γ, M)) ≡ $A \to B$ 对于一些A,B
　　　　和子类型(Γ,类型(Γ, N), A)然后B 否则失败
类型(Γ, $\lambda X<:A.M$)
　　如果好(Γ, A)然后 $\forall X<:A.$类型((Γ, $X<:A$), M)否则失败
类型(Γ, M A)
　　如果暴露(Γ,类型(Γ, M))≡ $\forall X<:A'.B$ 对于一些X,A',B
　　　　和好(Γ, A)和子类型(Γ, A, A')然后 $[A/X]B$ 否则失败

$F_{2<:}$提供了一个有趣的类型推断异常的例子。
上述的类型推断算法在理论上是不可判定的，但在实际中是可应用的。 它在几乎所有可能遇到的程序上都是收敛且高效的；只有在一些类型错误的程序上会发散，而这些程序应该被拒绝。 因此，$F_{2<:}$在可接受和不可接受类型系统之间处于边界位置，符合引言中的标准。

9 总结和研究问题

我们学到了什么

对于初学者程序员来说，自然的问题是：什么是错误？什么是类型安全？什么是类型完备性？ （也许可以这样问：计算机会告诉我哪些错误？为什么我的程序崩溃了？为什么计算机拒绝运行我的程序？） 即使是非正式的答案，也是非常复杂的。 我们特别关注类型安全和类型完备性之间的区别，并回顾了静态类型的各种变体。

在各种语言中，检查、动态检查和缺少检查程序错误是非常重要的。

从本章中最重要的教训是形式化类型系统的一般框架。 理解类型系统，从一般的角度来看，就像理解BNF（巴科斯-诺尔范式）一样重要：在没有精确的类型系统语言的情况下，很难讨论程序的类型；就像在没有精确的BNF语言的情况下，很难讨论程序的语法一样。在这两种情况下，形式主义的存在对于语言设计、编译器构建、语言学习和程序理解都有明显的好处。 我们描述了类型系统的形式主义，以及它如何捕捉类型的完整性和类型错误的概念。

借助形式化类型系统，我们开始描述一个广泛的程序构造列表及其类型规则。 这些构造中的许多是稍微抽象的版本，与熟悉的特性相似，而其他一些仅适用于常见语言的边缘领域。 在这两种情况下，我们的类型构造集合旨在作为解释编程语言的类型特性的关键。 这样的解释可能是非平凡的，特别是因为大多数语言定义不带有类型系统，但我们希望为独立学习提供足够的背景。 我们预计，一些高级类型构造将在未来的语言中更加完整、清晰和明确地出现。

在本章的后半部分，我们回顾了一些基本的类型推断算法：简单语言的算法、多态语言的算法以及具有子类型的语言的算法。 这些算法非常简单和通用，但大多数情况下只是具有说明性质。 由于众多实际原因，对于真实语言的类型推断变得更加复杂。然而，能够简洁地描述类型推断问题的核心及其一些解决方案是有趣的。

未来的方向

本章介绍的编程语言类型系统的形式化是类型理论的一种应用。 类型理论是形式逻辑的一个分支。 它旨在用有类型的逻辑取代谓词逻辑和集合论（它们是无类型的），作为数学的基础。

这些逻辑类型理论的动机之一，也是它们更令人兴奋的应用之一，是通过证明检查器和定理证明器对数学进行机械化。 类型在定理证明器中的应用与在编程中的应用原因相同，因此是有用的。 证明的机械化揭示了证明和程序之间的惊人相似之处：证明构造中的结构问题类似于程序构造中的问题。

证明需要有类型的编程语言的许多论证也证明了需要有类型的逻辑。

类型理论中开发的类型结构与编程中的类型结构的比较非常有教育意义。 函数类型、乘积类型、（不相交的）并集类型和量化类型在这两个领域中都有类似的用途。与常见编程语言的类型系统不同，集合论中使用的结构（如集合的并集和交集以及将函数编码为一对集合）在类型系统中没有对应物。

除了类型理论和编程之间最简单的对应关系之外，类型理论中发展的结构比编程中常用的结构更具表达力。 因此，类型理论为未来在编程语言中的进展提供了丰富的环境。

相反，程序员构建的系统规模远远大于数学家通常处理的证明规模。 大型程序的管理，特别是管理大型程序所需的类型结构，与管理机械证明相关。 在编程中开发的某些类型理论，例如面向对象和模块化的类型理论，超出了数学中常见的做法，并且对于证明的机械化有所贡献。

因此，逻辑和编程之间的相互交流将在类型理论的共同领域中继续进行。 目前，一些在编程中使用的高级构造逃脱了适当的类型理论形式化。 这可能是因为编程构造不合理，或者是因为我们的类型理论还不足够表达力：只有未来才能告诉我们。 活跃研究领域的例子包括高级面向对象和模块化构造的类型化以及并发和分布的类型化。

定义术语

抽象类型：一种数据类型，其本质被隐藏起来，只有预定的一组操作可以对其进行操作。

逆变：一种类型，其某个部分与其子类型相比，以相反的方向变化。 主要的例子是函数类型在其定义域中的逆变性。 例如，假设 $A<:B$并且将 X从 A变化到 B在 $X\to C$中；我们得到 $A\to C :> B\to C$。 因此，$X\to C$的变化方向与 X相反。

协变：一种类型，其某个部分与其子类型相比，以相同的方向变化。
例如，假设 $A<:B$并且将 X从 A变化到 B在 $D\to X$中；我们得到 $D\to A <: D\to B$。 因此，$D\to X$的变化方向与 X相同。

推导：通过应用类型系统的规则获得的判断树。

动态检查。一组运行时测试，旨在检测和防止禁止错误。

动态检查语言：在执行期间强制执行良好行为的语言。

显式类型语言：类型是语法的一部分的类型语言。

一阶类型系统：不包括对类型变量的量化的类型系统。

禁止错误：发生预定类别的执行错误之一；通常是将操作应用于值的不正确方式，例如不是(3)。

良好行为：与行为良好相同。

类型不正确：不符合给定类型系统规则的程序片段。

隐式类型语言：类型不是语法的一部分的类型语言。

判断：关于术语、类型和环境等实体的正式断言。 类型系统规定如何从其他有效的判断中产生有效的判断。

多态性：程序片段具有多个类型的能力（与单态性相反）。

安全语言：没有未捕获错误可能发生的语言。
二阶类型系统：包括对类型变量的量化，可以是普遍的或存在的。

静态检查。一组编译时测试，主要包括类型检查。
静态检查的语言：在执行之前确定良好行为的语言。

强检查的语言：在运行时不会发生禁止的错误（取决于禁止错误的定义）。

包含：子类型的基本规则，断言如果一个术语具有类型 A，它是类型 B的子类型，那么该术语也具有类型 B。
子类型：一种自反且传递的二元关系，满足包含关系；它断言了值的集合的包含关系。
陷入错误：立即导致故障的执行错误。
类型：一组值的集合。 程序片段在程序执行期间可以假定的值的估计集合。

类型推断：在给定类型系统中为程序找到类型的过程。
类型重构：在给定类型系统中找到省略了类型信息的程序的类型的过程。

类型规则：类型系统的一个组成部分。 规定了特定程序结构不会引起禁止错误的条件的规则。
类型安全：指程序不会引起未被捕获的错误的属性。
类型完备性：指程序不会引起禁止错误的属性。
类型系统：一种带有类型规则的类型化编程语言的集合。 与静态类型系统相同。

类型检查器：编译器或解释器的一部分，执行类型检查。
类型检查：在执行之前检查程序以确保其符合给定的类型系统，从而防止出现禁止的错误。
类型化语言：具有关联（静态）类型系统的语言，无论类型是否是语法的一部分。

类型错误：由类型检查器报告的错误，用于警告可能的执行错误。
未捕获错误：不会立即导致故障的执行错误。
无类型语言：没有（静态）类型系统的语言，或者其类型系统具有包含所有值的单一类型。
有效判断：从给定类型系统的推导中获得的判断。
弱检查语言：静态检查但没有明确保证不存在执行错误的语言。

良好行为：在运行时不会产生禁止的错误的程序片段。
良好形式：根据形式规则正确构造的。
良好类型化程序：符合给定类型系统规则的程序（片段）。

参考文献

[1] Aiken, A.和E.L. Wimmers,类型包含约束和类型推断,*Proc. ACM* 函数式编程和计算机体系结构会议, 31-41. 1993年。

[2] Amadio, R.M.和L. Cardelli,子类型递归类型。 ACM程序设计语言和系统交易 **15**(4), 575-631. 1993年。

[3] Birtwistle, G.M.、O.-J. Dahl、B. Myhrhaug和K. Nygaard, **Simula入门** . Studentlitteratur. 1979年。

[4] Böhm, C.和A. Berarducci,在术语代数上自动合成带类型的 λ-程序. 理论计算机科学 **39**, 135-154. 1985年。

[5] Cardelli, L.,基本多态类型检查.计算机编程科学 **8**(2). 147-172. 1987年。

[6] Cardelli, L.,在子类型的纯演算中的可扩展记录. 在面向对象编程的理论方面,C.*A*. G**unter**和J.C. M**itc***h***ell**, 编辑. MI*T*出版社. 373-425. 1994年.

[7] Cardelli, L.和P. Wegner,关于理解类型、数据抽象和多态. *ACM*计算调查 **17**(4), 471-522. 1985年.

[8] Curien, P.-L.和G. Ghelli,在$F_{\leq}$中的包含性、最小类型和类型检查的一致性.计算机科学中的数学结构 **2**(1), 55-91. 1992年.

[9] Dahl, O.-J., E.W. Dijkstra和C.A.R. Hoare,结构化编程. 学术出版社, 1972年.

[10] Eifrig, J., S. Smith和V. Trifonov,对象的声音多态类型推断. *Proc. OOPSLA'95*, 169-184. 1995年.

[11] Gunter, C.A.,编程语言的语义：结构和技术. 计算机基础, M. G**are**y 和 A. M**eye**r 编. MIT P**ress**. 1992.

[12] Girard, J.-Y., Y. Lafont, 和 P. Taylor, 证明和类型 . Cambridge University Press. 1989.

[13] Gunter, C.A. 和 J.C. Mitchell, 编.,面向对象编程的理论方面. MIT Press. 1994.

[14] Huet, G. 编.,函数式编程的逻辑基础. Addison-Wesley. 1990.

[15] Jensen, K.,**Pascal** 用户手册和报告，第二版. *Springer Verlag*, 1978.

[16] Liskov, B.H.,**CLU** 参考手册. 计算机科学讲义 114. Springer-Verlag. 1981.

[17] 米尔纳，R.，编程中的类型多态性理论. 计算机和系统科学杂志 **17**, 348-375. 1978.

[18] 米尔纳，R.，M. 托夫特和R. 哈珀，标准M**L**的定义. MIT出版社. 1989.

[19] 米切尔，J.C.，强制和类型推断.第11届*A*CM编程语言原理研讨会论文集, 175-185. 1984.

[20] 米切尔，J.C.，编程语言的类型系统. 在理论计算机科学手册中，J. v**an** Lee**u**we**n**编辑. 北荷兰. 365-458. 1990.

[21] 米切尔，J.C.:编程语言基础. MIT出版社, 1996.

[22] 米切尔，J.C.和G.D. Plotkin，抽象类型具有存在类型.第*12*届**A**CM编程语言原理研讨会论文集. 37-51. 1985.

[23] Nordström, B., K. Petersson, and J.M. Smith,**Martin-Löf**类型理论中的编程. 牛津科学出版社。1990年。

[24] Palsberg, J.,对象类型的高效推断.信息与计算。 **123**(2), 198-209. 1995年。

[25] Pierce, B.C.,**有界量化是不可判定的.第*19*届*ACM*编程语言原理研讨会**. 305-3*15*. *1992***年。**

[26] Pierce, B.C.,**类型与编程语言**. MIT Press, 2002**年。**

[27] Reynolds, J.C.,向类型结构理论迈进.*Colloquium sur la programmation*, 408-423. 计算机科学讲义 19. Springer-Verlag. 1974年。

[28] **雷诺兹，**J.C.**，类型，抽象和参数多态. 在信息处理**-中，R.E.A.梅森，编。北荷兰。513-523。1983年。

[29] **施密特，**D.A.**，类型化编程语言的结构.麻省理工学院出版社。**1994**年。**

[30] **斯宾塞，**H.**，C程序员的十诫（注释版）. 可在**全球网络上获得。

[31] **托夫特，**M.**，多态引用的类型推断.信息与计算 89**, 1-34。1990年。

[32] **韦尔斯，**J.B.**，第二阶 λ-演算中的可类型化和类型检查是等价的**和不可判定的。第*9*届*IEEE*计算机科学逻辑学研讨会论文集, 176-185。1994.

[33] **温加登，**V.**，编。Algol68算法语言修订报告**. 1976**年。**

[34] **莱特**, A.K. **和** M. **费莱森**,**一种语法方法来保证类型的完整性**. **信息与计算 115**(1), 38-94. 1994.

进一步的信息

要全面了解类型系统的背景，应该阅读(1)一些关于类型理论的材料，通常比较难懂，(2)一些将类型理论与计算机联系起来的材料，以及(3)一些关于具有高级类型系统的编程语言的材料。

由Huet [14] 编辑的书涵盖了类型理论的各种主题，包括几篇教程文章。 由Gunter和Mitchell [13] 编辑的书收集了一些关于面向对象类型理论的论文。 由Nordström、Petersson和Smith [23] 编写的书是对Martin-Löf的工作的最新总结。 Martin-Löf提出了类型理论作为一种根植于计算的通用逻辑。 他引入了本章中使用的判断和类型规则的系统化符号表示法。 Girard和Reynolds [12, 27] 发展了多态 λ-演算(F_2)，这激发了本章所涵盖的许多工作的灵感。

从类型系统研究中产生的技术问题的现代阐述可以在皮尔斯的书[26]、冈特的书[11]、米切尔在《理论计算机科学手册》[20]中的文章以及米切尔的书[21]中找到。

接近编程语言，丰富的类型系统在Algol的发展和结构化编程的建立之间的时期得到了开创性的发展[9]，并发展成了一代新的丰富类型语言，包括Pascal [15]、Algol68 [33]、Simula [3]、CLU [16]和ML [18]。 Reynolds对多态性和数据抽象给出了类型理论的解释[27, 28]。 关于这个主题，还可以参考[7, 22]。Schmidt的书[29]涵盖了本章讨论的几个问题，并提供了更多关于常见语言构造的细节。

Milner关于ML的类型推断的论文[17]将类型系统和类型推断的研究推向了一个新的水平。 它包括了一种多态类型推断算法，并且基于一种指示性技术给出了（简化的）编程语言的类型完备性的第一个证明。

独特。 在[5]中可以找到更易理解的算法描述。
类型安全性的证明现在通常基于操作技术[31, 34]。 目前，标准ML是唯一一个广泛使用的具有正式指定类型系统的编程语言[18]，尽管类似的工作现在已经针对Java的大片段进行了。