第19章 类型继承

19.1 引言

在第13章涉及了子类型和超类型的概念，更确切地说是实体子类型和实体超类型。在该章中指出，假设某些雇员是程序员，而所有的程序员都是雇员，那么可以认为实体类型PROGRAMMER（程序员）是实体类型EMPLOYEE（雇员）的子类型，而EMPLOYEE是PROGRAMMER的超类型。同时也指出，一个“实体类型”并不是一个非常规范意义上的“类型”（部分原因是由于“实体”本身没有正式的定义）。在本章里，我们将深入地分析子类型和超类型。而对于“类型”将使用第5章中给出的比较规范和准确的定义。因此，先从更准确的“类型”定义开始：

- 一个类型是一组值的命名集合（即被讨论类型的所有可能的取值），以及与之相关的可以在属于该类型的值或变量上应用的操作的集合。

进一步地讲：

- 任何给定的类型可以是系统定义的或是用户定义的。
- 任何给定类型的定义是对该类型中所有有效值构成的集合的规格说明（规格说明就是在第8章所说的有效类型约束）。
- 这些值可以具有任意的复杂度。
- 这些值的实际表示形式或物理表示形式对用户而言通常是不可见的，即类型与其（实际）表示形式是不同的。但是每种类型至少要通过合适的THE_操作（或是其他等价的逻辑操作）明确地给用户提供一种合适的表示形式。
- 任何给定类型的值或变量只能通过定义在该类型上的操作进行操作。
- 除了已经提到的THE_操作，类型的操作还包括：
  a. 至少要有一个选择子操作（更确切地说，对应于每种可能的外在表示形式都应该有一个这样的操作），通过合适的选择子操作调用可以实现对类型中的每个值进行选取和引用。
  b. 一个相等操作，可以检查相同类型的两个值是否相等。
  c. 一个赋值操作，可以将一个值赋给同类型的一个变量。
  d. 一些类型检测操作，我们将在19.6节讨论它们。注意：对于不支持继承的情况，这些操作可能是不需要的。

在上述的基础上，现在指出：

- 某些类型是另外一些超类型的子类型。如果*B*是*A*的子类型，则所有适用于*A*的操作和类型约束都适用于*B*（继承），但同时*B*拥有属于自己的、并不适用于*A*的操作和类型约束。

举个例子，假设有ELLIPSE（椭圆）和CIRCLE（圆）两种类型。顾名思义，可以说CIRCLE是ELLIPSE的子类型（ELLIPSE是CIRCLE的超类型）。这实际上是说：

- 每个圆同时也是一个椭圆（即所有圆的集合是所有椭圆的集合的一个子集），但是反之

并不成立。

- 因此，普遍适用于椭圆的操作也相应地适用于圆（因为圆属于椭圆），但是反之并不成立。比如，操作 THE_CTR（取中心）可以适用于椭圆，从而也适用于圆，但是操作 THE_R（取半径）则只适用于圆。
- 此外，普遍适用于椭圆的约束也相应地适用于圆（同样是由于圆属于椭圆），但是反之并不成立。比如，椭圆遵循约束 $a \geq b$（a和b分别是椭圆的长半轴和短半轴），则圆必定也满足这一约束。当然对于圆而言，a、b重合于半径r，但约束还是得到了满足。事实上确切地说，约束 $a=b$只适用于圆而并不能普遍适用于椭圆。注意：在本章里，我们所说的“约束”一词是指一种类型约束，所说的“半径”和“半轴”实际上是指相应的半径长度和半轴长度。

小结一下：简单地说，类型 CIRCLE继承了类型ELLIPSE的操作和约束，同时还有属于自己的操作和约束，但是这些操作和约束并不适用于类型 ELLIPSE。于是这样一种情况往往容易引起概念上的混淆，即子类型同时具有超类型的值的一个子集和属性的一个扩展集。注意：在整个这一章里，我们使用“属性”这个词来作为“操作和约束”的简称。

1. 为什么要讨论类型继承

为什么这个问题值得探讨呢？这至少有两点原因。

- 子类型与继承的概念在现实世界中是自然存在的，即这种情形还是会经常碰到的。一个给定类型的所有值具有某些共同的属性，而这些值的某些子集具有更多属于它们自己的特定的属性。这样，子类型与继承看起来是“为现实建模”（modeling reality）（或者像在第13章所说的，是“语义建模”(semantic modeling）的一个有用的工具。
- 第二，如果能够识别出这些模式，即子类型和继承的模式，并能建立起识别方法，把这些模式纳入到应用软件和系统软件中，我们也许可以获得某些实在的好处。比如，一个可以应用于椭圆的程序或许也可以应用于圆，哪怕在编写程序的时候根本就没有考虑过关于圆的问题（也许在编写程序的时候类型 CIRCLE还没有定义），这种好处就是所谓的代码重用。

尽管有上面所说的这些潜在的好处，但是迄今似乎还没有对于一个规范的、严格的、抽象的类型继承模型达成任何一致意见。引用文献 [19.10]的话说就是：

> 继承的基本概念非常简单……（而且，尽管）它在现有系统中所处的中心地位，继承仍然是一种有争议的机制……（一个）全面的关于继承的概念还是没有。

本章所做的讨论是基于作者与 Hugh Darwen共同建立的模型，该模型在文献 [3.3]中有详尽的描述。因此必须明确，其他作者和其他文章在使用诸如“子类型”和“继承”等概念的时候，其使用角度可能与我们所讨论的角度并不一样。

2. 预备知识

在逐节正式讨论继承之前，先要澄清一些基本概念，这些概念是本小节的主题。

- 值是有类型的

在第5章中曾说过，如果v是一个值，则可以认为v带有某种标志来表明“我是一个整数”或“我是一个供应商号”或“我是一个圆”，等等。如果不存在继承，那么一个值只属于一种类型。但是如果存在继承，那么一个值就可以同时属于多个类型。例如，一个给定值可以同时属于类型 ELLIPSE和CIRCLE。

- **变量是有类型的**

  每个变量都有一个声明的类型，比如可以声明如下一个变量：

  ```
  VAR E ELLIPSE;
  ```

  这里变量E的类型声明为ELLIPSE。如果不存在继承，一个给定变量所能取的值就只属于一个类型，即该变量的声明类型。但是，如果存在继承，一个变量所具有的值就可能同时属于多个类型。比如，变量E的当前值可能是一个圆（同时也是一个椭圆），因此这个值就同时属于类型ELLIPSE和CIRCLE。

- **单一继承与多重继承**

  类型继承主要有两种情况：单一继承与多重继承。简单地说，单一继承是指每个子类型只有一个超类型，从而只继承一种类型的属性；多重继承是指一个子类型可以有多个超类型，并继承了所有这些类型的属性。显而易见，前一种情况是后一种情况的特例。可是即便是单一继承就已经很复杂了（虽然这有些让人吃惊，但事实上确实如此），因此在本章里，我们只把注意力放在单一继承上，我们所说的继承都是特指单一继承。关于单一继承和多重继承的详细讨论请见文献[3.3]。

- **标量、元组和关系继承**

  很明显继承既包括标量值也包括非标量值[⊖]，因为那些非标量值最终是由标量值构成的。当然特别地，继承还包括针对元组值和关系值的继承。但是，光是标量继承就已经相当复杂了，因此在本章里我们只把注意力集中在标量继承上，而我们所说的类型、值以及变量实际上都是指标量类型、标量值和标量变量。在文献[3.3]中有关于各种继承的详细讨论，包括标量继承、元组继承和关系继承。

- **结构继承与行为继承**

  标量值可以拥有一个具有任意复杂度的内部（物理）结构或表现形式。例如，椭圆和圆在适当的情形下（这种情形我们已经知道），都可以很自然地被看作是标量值，即使它们的内部结构可能非常复杂。但是，其内部结构通常对于用户是不可见的。从而当谈到继承时（至少是对于模型而言），其中并不包括结构的继承，因为从用户的角度来看并没有结构需要继承。换句话说，感兴趣的是所谓的行为继承，而不是结构继承（这里的“行为”是指操作，虽然约束也可以继承，至少在模型中是这样）。注意：当然并不排除结构继承，只是把它看作是一个实现上的问题，从而与我们的模型无关。

- **“子表与父表”**

  现在应该清楚了，继承模型所关注的是关系术语中所说的域继承（再次提醒，域和类型是相同的东西）。但是在关系范畴里谈到继承的可能性时，大部分人立刻会想到是在讨论某一类的表继承。例如，SQL3标准支持所谓的“子表与父表”，据此标准，*B*可以在继承表*A*的所有列之后再加上一些自己的列（见附录B）。而在我们看来，“子表与父表”的概念是一个完全独立的现象，虽然它同时可能也是很有意义的（尽管我们在文献[13.12]中对此表示了怀疑），但是它在本质上与类型继承毫无关系。

  最后一个要说明的基本概念是：类型继承是一个与所有普遍意义上的数据都有关的问题，

⊖ 要记住，标量类型没有任何用户可见的分量。像第5章所说的，不要因为标量类型的某些可能的表示形式最终会有显式的分量而产生误解。这些分量是某种表示形式的分量而不是类型的分量——尽管有时我们会有些随便地引用它们，使得它们看起来好像确实是类型的分量。

而不是仅仅局限于数据库中的数据。因此，为了简单起见，本章的大部分例子是以局部数据（普通程序变量，等等）的形式表达的，而不是数据库中的数据。

19.2 类型的层次结构

先给出一个在本章里要用到的例子。这个例子包括一组几何类型——PLANE_FIGURE（平面图形）、ELLIPSE、CIRCLE、POLYGON（多边形），等等，这些类型组织为一个所谓的类型层次（type hierarchy），或者更一般地讲，一个类型图（type graph）（见图19-1）。这里列出的是Tutorial D中对于其中一些几何类型的定义（请特别注意其中的类型约束）：

```
TYPE PLANE_FIGURE ... ;
TYPE ELLIPSE POSSREP ( A LENGTH, B LENGTH, CTR POINT )
     SUBTYPE_OF ( PLANE_FIGURE )
     CONSTRAINT ( THE_A ( ELLIPSE ) ≥ THE_B ( ELLIPSE ) ) ;
TYPE CIRCLE POSSREP ( R LENGTH, CTR POINT )
     SUBTYPE_OF ( ELLIPSE )
     CONSTRAINT ( THE_A (CIRCLE ) = THE_B ( CIRCLE ) ) ;
```

这样系统就会知道CIRCLE是ELLIPSE的子类型，因此，普遍适用于椭圆的操作和约束也就相应适用于圆。

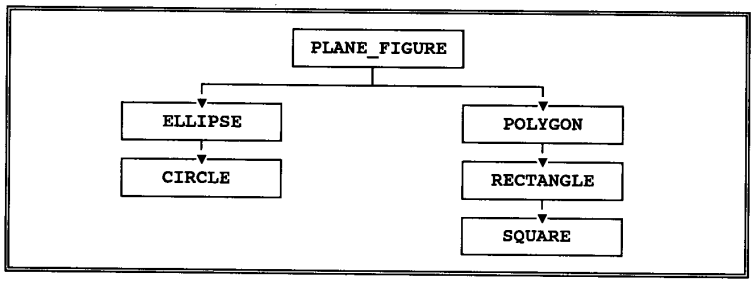

图19-1 一个简单的类型层次结构

主要详细阐述一下类型ELLIPSE和CIRCLE的POSSREP的说明。首先，为了简单起见，假设椭圆总是长轴处在水平而短轴处在垂直的位置状态下，这样椭圆可以用它们的半轴 a、b（包括中心）来表示。相应地，圆可以用它们的半径（加上中心）来表示。与第8章一样，我们还假设椭圆的长半轴 a 总是大于等于它们的短半轴 b 的（即它们是“矮而胖”的，而不是“高而瘦”的）。

下面列出的是与上述类型有关的一些操作的定义。

```
OPERATOR AREA ( E ELLIPSE ) RETURNS ( AREA ) ;
    /*“面积”——注意AREA既是操作本身的名字又是结果的类型的名字*/
END OPERATOR ;
OPERATOR THE_A ( E ELLIPSE ) RETURNS ( LENGTH ) ;
    /*“半轴a的长度”*/
END OPERATOR ;
OPERATOR THE_B ( E ELLIPSE ) RETURNS ( LENGTH ) ;
    /*“半轴b的长度”*/
END OPERATOR ;
OPERATOR THE_CTR ( E ELLIPSE ) RETURNS ( POINT ) ;
    /*“中心”*/
```

```
END OPERATOR ;
OPERATOR THE_R ( C CIRCLE ) RETURNS ( LENGTH ) ;
    /*“半径长度”*/
END OPERATOR ;
```

除了THE_R以外，所有这些操作都适用于属于 ELLIPSE类型的值，因此也毋庸置疑同样适用于属于CIRCLE类型的值。而相应地，THE_R操作就只适用于属于CIRCLE类型的值。

1. 术语

在继续下面的讨论之前，我们还需要介绍一些定义和名词。但是这些概念是非常浅显的。

1) 超类型的超类型本身还是超类型。比如， POLYGON是SQUARE（正方形）的一个超类型。
2) 每个类型都是自己的一个超类型。比如， ELLIPSE也是ELLIPSE的超类型。
3) 如果*A*是*B*的超类型，而*B*是不同于*A*的类型，则*A*是*B*的一个真（proper）超类型。比如，POLYGON是SQUARE的真超类型。

当然，类似的说法也适用于子类型。即：

4) 子类型的子类型还是子类型。比如， SQUARE是POLYGON的一个子类型。
5) 每个类型都是自己的一个子类型。比如， ELLIPSE也是ELLIPSE的子类型。
6) 如果 *B*是*A*的子类型，而 *A*是不同于 *B*的类型，则 *B*是*A*的一个真子类型。比如，SQUARE是POLYGON的真子类型。

还有：

7) 如果*A*是*B*的超类型，而且不存在一个类型 *C*既是*A*的真子类型又是 *B*的真超类型，则 *A*是 *B* 的一个 直接 （immediate）超类型，而 *B*是*A*的一个直接子类型。比如，RECTANGLE（矩形）是SQUARE的直接超类型，同时 SQUARE是RECTANGLE的直接子类型（因此要注意，在我们的 Tutorial D的语法中，关键字SUBTYPE_OF的意思是特指“是……的直接子类型”）。
8) 根（root）类型是没有超类型的类型。比如， PLANE_FIGURE就是一个根类型。注意：我们并没有假设只有一个根类型。但是，如果有两个或多个根类型，我们总可以构造出一个“系统”类型做为所有这些类型的直接超类型，所以假设只有一个根类型并不会丧失普遍性。
9) 叶（leaf）类型是没有子类型的类型。比如， CIRCLE就是一个叶类型。注意：这样一个定义是有些简化了的，但是对于目前的讨论是足够了（要完全适合多重继承，它还需要进行一些小小的扩充[3.3]）。
10) 每个真子类型只有一个直接超类型。注意：这里要明确的是我们的假设只是针对单一继承的。如前所述，放宽假设的情况在文献 [3.3]中有详细讨论。
11) 只要(a)至少存在一个类型；而且 (b)不存在回路 —— 即不存在一系列的类型 *T1*、*T2*、*T3*、...、*Tn*，使得*T1*是*T2*的直接超类型， *T2*是*T3*的直接超类型，……，而 *Tn*是*T1*的直接超类型 —— 则至少有一个类型必定是根类型。注意：事实上，是不可能存在任何回路的（为什么不可能呢？）。

2. 不相交假设

再做如下一个简化假设：如果 *T1*和*T2*是不同的根类型，或者是同一超类型的不同直接子

类型（尤其是指任何一个都不是另外一个的子类型），则假设它们是不相交的——即没有既属于类型*T1*又属于类型*T2*的值。例如，没有既属于椭圆又属于多边形的值存在。

下列进一步的观点是这一假设的直接推论：

12) 不同的类型层次结构之间是不相交的。

13) 不同的叶类型之间是不相交的。

14) 每个值都只有一个最确切（most specific）的类型。比如，一个给定值也许“只是一个椭圆”而不是一个圆，这就是说，它的最确切的类型是ELLIPSE（在现实世界中，很多椭圆都不是圆）。实际上，更严格地说，如果某个值*v*的最确切类型是*T*，则指*v*所属的类型的集合就是*T*的所有超类型的集合（当然其中也包括*T*本身）。

需要不相交假设的一个原因是它可以避免有可能出现的多义性。假设某个值*v*同时属于类型*T1*和*T2*，两者都不是对方的子类型。进一步假设*T1*定义了一个叫*Op*的操作，而*T2*也定义了一个叫*Op*的操作[⊖]，则对*v*应用*Op*操作就会产生二义性。

注意：不相交假设在我们只把注意力局限于单一继承的时候才是合理的，但是对于多重继承的情况这一假设必须放宽。具体的讨论可以参看文献[3.3]。

3. 关于物理表示形式的一点说明

虽然我们主要关心的是继承的模型而不是继承的实现问题，但是为了能够对继承的整体概念有一个正确的理解，必须对于特定的实现问题有一定程度的了解。下面就进行这样一点说明：

15) *B*是*A*的子类型这一事实并不能说明，*B*的值的实际表示形式（对用户不可见）与*A*的值是一样的[⊜]。比如，椭圆可能实际上是用其中心和半轴来表示的，而圆可能是用其圆心和半径来表示的（虽然一般而言，实际的表示形式并不需要和任何已知的可能表示形式一样）。

你将会看到这一点在下面的许多小节中都会显得相当重要。

19.3 多态性和可置换性

这一节研究两个非常重要的概念：多态性和可置换性。由这两者共同构成的基础，可以获得在19.1节中提到的代码重用带来的好处。还必须立即明确一点，即这两个概念实际上是从不同的角度看待同一件事情。尽管如此，还是让我们先来看看多态性。

1. 多态性

恰恰是继承的概念意味着，如果*T'*是*T*的一个子类型，则所有适用于*T*的值的操作也同样适用于*T'*的值。例如，如果AREA(*e*)是合法的，这里*e*是一个椭圆，并且如果*c*是一个圆，则AREA(*c*)也是合法的。由此就要非常注意一个给定操作的参变量（parameter）以及它们的声明类型与调用该操作时的相应参数（argument）以及它们的实际（最确切）类型之间的差异。

⊖ 换句话说，*Op*是一个多态操作。而多态可以有重载(overloading)多态和包含(inclusion)多态。在19.3节中会又进一步的说明。

⊜ 事实上，在逻辑上并没有任何理由说明同一类型的值必须有同样的实际表示形式。举例来说，有些点可能用笛卡尔坐标来表示，而有些点则可能用极坐标来表示；有些温度可能用摄氏度来表示，而有些温度则可能用华氏度的形式；有些整数可能使用十进制表示，而有些整数则可能使用二进制，等等（当然，系统必须知道在这些情况下如何在这些实际表示形式之间进行转化，以便正确地进行赋值、比较等操作）。

比如，操作AREA是通过一个声明类型为ELLIPSE的参变量来定义的（见19.2节），但是在调用中，AREA(*c*)的参数的实际（最确切）类型是CIRCLE。

现在再次重申椭圆和圆具有不同的可能表示形式，至少在19.2节中是这样定义的：

```
TYPE ELLIPSE POSSREP ( A LENGTH, B LENGTH, CTR POINT ) ... ;
TYPE CIRCLE POSSREP ( R LENGTH, CTR POINT ) ... ;
```

因此，很有可能在同样的形式下，AREA操作存在两个不同的版本，一个针对ELLIPSE类型的可能表示形式，一个针对CIRCLE类型的可能表示形式。但是仅仅是很有可能——而不是必须。比如，ELLIPSE类型可以有这样的代码：

```
OPERATOR AREA ( E ELLIPSE ) RETURNS ( AREA ) ;
RETURN ( 3.14159 * THE_A ( E ) * THE_B ( E ) ) ;
END OPERATOR ;
```

（椭圆的面积是πab）。显然，当由一个圆而不是一个普通的椭圆来调用的时候，这样的代码也能得到正确的结果，因为对于一个圆来说，操作THE_A和THE_B返回的都是半径*r*。但是，也许由于各种各样的原因，定义类型CIRCLE的人也许更愿意专门针对圆实现AREA的一个版本，而通过调用THE_R来代替对THE_A和THE_B的调用。注意：事实上就算是可能的表示形式相同，但是出于效率上的考虑，也可能需要对同一个操作实现两个不同的版本。就拿多边形和矩形来说，计算一个普通多边形面积的算法肯定也适用于一个矩形，但是对于矩形而言还有一个更有效的算法——长乘宽。

但是也应该注意到，如果ELLIPSE的代码是针对ELLIPSE的实际表示形式而不是可能的表示形式来写的，那么这些代码可能真的无法适用于圆，因为ELLIPSE和CIRCLE的实际表示形式是不同的。通常针对实际的表示形式实现操作并不是一个好主意，编制代码要保守一些。

无论怎样，如果不需要为CIRCLE重写AREA的代码，我们就实现了代码重用（对AREA的实现而言）。注意：在下一小节中我们还将遇到一种更重要的“重用”。

当然从模型的角度来看，AREA到底有多少个版本并不重要（就用户而言，在定义中只有一个AREA操作，它既适用于椭圆也适用于圆）。换句话说，从模型的角度而言，AREA是多态的：在不同的调用中它可以引用不同类型的参数。因此必须切实地注意到，这种多态性是继承的逻辑推论：如果采用继承，就必须接受多态性，否则就不能采用继承。

现在也许已经意识到多态性本身并不是一个新的概念。比如，SQL语言中就有多态操作（“=”、“+”、“||”等等），事实上大部分其他程序设计语言也是这样的。有些语言甚至允许用户定义自己的多态操作。比如PL/I就通过所谓的“GENERIC函数”提供这样的功能。但是在如上所述的各个例子中并没有实现任何的继承，它们都是通常所说的重载多态的例子。而相应地，操作AREA所表现的多态叫做包含多态，这是因为椭圆与圆之间的关系主要是集合之间的包含关系[19.3]。由于很明显的原因，在本章的剩下部分里，我们都用“多态”来表示包含多态，而不使用明确的表述形式。

注意：下面的说明也许有助于了解重载多态与包含多态之间的区别：

- 重载多态是指许多不同的操作具有相同的名字（而且用户并不需要知道这些操作事实上是不同的，同时还有不同的语义——虽然如果语义相似更好）。比如，“+”在大部分语言中都是重载的（一个操作“+”用于整数相加，而另一个操作“+”用于有理数相加，等等）。

• 包含多态是指只有一个操作，但是在同一个形式下有多个不同的实现版本（但是用户并不需要知道是否真的有多个实现版本——重复一遍，对用户而言只知道只有一个操作）。

2. 利用多态编程

来看看下面这个例子。设想我们要写一个程序来显示一些由正方形、椭圆以及圆等构成的图形。如果不用多态，代码可能与下面的伪代码类似：

```
FOR EACH x IN DIAGRAM
CASE ;
   WHEN IS_SQUARE ( x ) THEN CALL DISPLAY_SQUARE ... ;
   WHEN IS_CIRCLE ( x ) THEN CALL DISPLAY_CIRCLE ... ;
          ......
END CASE;
```

（假设有诸如IS_SQUARE、IS_CIRCLE这样的操作来检查一个给定的值是否属于特定的类型）。相应地，如果采用多态，代码就会非常的简洁明了：

```
FOR EACH x IN DIAGRAM CALL DISPLAY x () ;
```

说明：DISPLAY在这里是一个多态操作。针对不同类型 *T*的值，DISPLAY的实现版本会在定义类型*T*的时候相应地进行定义，并同时告诉系统。这样在运行的时候，如果系统遇到了带参数*x*的DISPLAY调用，它必须识别出 *x*的最确切类型，然后调用适合该类型的 DISPLAY版本——这是一个动态联编（run-time building）的过程 ㊀。换句话说，多态实际上是把原来出现在用户源代码中的CASE表达式和CASE语句转移到了统一的表示形式下：系统实际上代替用户执行了CASE操作。

让我们再来着重看看维护上述程序意味着什么。举个例子，假如又定义了一个新类型TRIANGLE（三角形）做为 POLYGON的另外一个直接子类型，那么现在显示的图形就又可以包括三角形了。如果不用多态，则每一个如同上面一样包含 CASE表达式或CASE语句的程序现在都必须进行修改，加上如下形式的代码

```
WHEN IS_TRIANGLE ( x ) THEN CALL DISPLAY_TRIANGLE ... ;
```

而如果采用多态，这样的源代码修改就不需要了。

由于类似上述例子的众多例子，有时候多态性会被形象地说成是“让旧代码调用新代码”，即一个程序*P*可以有效地调用某些操作的某些版本，即使是操作的这些版本在编写程序的时候是没有的。所以现在我们有了另外一个——同时也是更为重要的——代码重用的例子：即使在编写程序*P*时类型*T*并不存在，*P*仍然可以应用在属于*T*的数据上。

3. 可置换性

就像在前面提到的，可置换性实际上是从一个稍微有些不同的角度来看待多态性这一概念。比如，我们已经看到，当 *e*为椭圆时，如果AREA(*e*)是合法的，则当*c*为圆时，AREA(*c*)也是合法的。换句话说，在系统中任何可以接受椭圆的地方，都可以用圆来置换。更一般地，在系统中任何可以接受属于类型 *T*的值的地方，都可以用属于类型 *T*′的值来置换，这里的*T*′是*T*的子类型。这就是值的可置换性原则。

特别要注意的是在这一原则下，如果某个关系 *r*的属性*A*声明为ELLIPSE类型，则*r*中*A*的

㊀ 动态联编当然也是一个实现问题而不是一个模型问题。但它也是一个需要有一定了解的实现问题，从而有助于对继承的整体概念有正确的理解。

值可以属于类型CIRCLE而不仅仅是类型ELLIPSE。类似地，如果某个类型*T*的一个可能的表示形式中有一个声明为ELLIPSE类型的分量*C*，那么对于属于类型*T*的某些值*v*而言，执行操作THE_C(*v*)所得到的返回值可能属于类型CIRCLE而不仅仅是类型ELLIPSE。

最后，我们注意到，既然可置换性是多态性的另一种表示，则它同样也是继承的逻辑推论：如果有继承存在，就必定要接受可置换性，否则就没有继承。

19.4 变量与赋值

假设有两个变量E和C分别声明为类型ELLIPSE和CIRCLE。

```
VAR E ELLIPSE ;
VAR C CIRCLE ;
```

首先我们用某个圆初始化*C*——比如（为了确切起见）是一个半径为3、中心在原点的圆：

```
C := CIRCLE ( LENGTH ( 3.0 ), POINT ( 0.0, 0.0 ) ) ;
```

表达式右边是对类型CIRCLE的选择子操作的调用（在第5章中曾说过，对于每一种声明的可能表示形式都有一个同名的选择子操作，该操作的参变量对应于该表示形式的各个分量。选择子操作的作用是允许用户通过给相应的表示形式的每个分量赋值，来指定或者说“选择”一个属于该类型的值）。

现在考虑如下的赋值：

```
E := C ;
```

一般情况下——即不存在子类型和继承的情况下——赋值操作要求表达式左边的变量与表达式右边指定的值具有相同的类型（对于变量而言是具有相同的声明类型）。但是值的可置换性原则表明在系统中任何可以接受属于类型ELLIPSE的值的地方，都可以用一个CIRCLE类型的值来置换，所以上面的赋值是合法的（实际上“赋值”是一个多态操作）。其效果是从变量C赋一个圆的值到变量E中，而且，赋值后变量E的值的类型是CIRCLE而不仅仅是ELLIPSE。换句话说：

- 在给一个声明类型为非确切（less specific）类型的变量赋值时，值可以保持其自身的最确切类型。在这类赋值中并不出现类型转换（在这个例子中，圆并不被转换成一个椭圆）。注意，我们实际上并不希望出现任何类似的转换，因为这样会让所赋的值丢失其最确切行为，对这个例子而言，就是在赋值之后我们将无法得到变量E中圆的半径。注意：在本节后面的“类型下移”这一小节中将讨论取得半径所涉及的问题。
- 由此可见，可置换性暗示着一个声明为类型*T*的变量可以具有任何值，只要这个值的最确切类型为*T*的任意一个子类型。因此，必须仔细区分一个变量的声明类型和该变量（当前值）的实际类型——即最确切类型——之间的区别。在下一小节中我们还要对这个重要问题进行讨论。

接下来在这个例子中，假设有另外一个声明为AREA类型的变量A：

```
VAR A AREA ;
```

考虑如下的赋值：

```
A := AREA ( E ) ;
```

将会产生如下后果：

- 首先，系统将对表达式AREA(E)执行编译时的类型检查。由于E的声明类型为ELLIPSE，

同时操作AREA的唯一参变量的声明类型也是 ELLIPSE（见19.2）节，类型检查可以通过。

- 其次，在运行的时候系统发现 E的当前最确切类型是 CIRCLE，因此调用适用于圆的AREA的版本（换句话说，系统执行了在前面一节讨论过的动态联编过程）。

当然，系统实际调用的是AREA的圆的版本而不是椭圆的版本，对用户而言是没有意义的——再次说明，对于用户而言只有一个 AREA操作。

1. 标量变量

对于声明类型为 *T*的标量变量 *V*来说，其当前值 *v*的最确切类型可以是 *T*的任意子类型。由此可见，可以用一个形如 <*DT*,*MST*,*v*>的有序三元组来把 *V*模型化（我们也确实是这样做的），其中：

- *DT*是变量*V*的声明类型。
- *MST*是变量*V*的当前最确切类型。
- *v*是一个属于最确切类型 *MST*的值 —— 也就是变量 *V*的当前值。

用符号 *DT*(*V*)、*MST*(*V*)和*v*(*V*)分别代表标量变量 *V*的声明类型、当前最确切类型和当前值。注意到(a)*MST*(*V*)总是*DT*(*V*)的子类型 —— 当然不必是真子类型；(b)一般来说，*MST*(*V*)和*v*(*V*)是随着时间变化的；(c)实际上*MST*(*V*)是包含在*v*(*V*)中的，因为每个值只有一个最确切类型。

标量变量的这个模型对于明确各种操作的精确语义是非常有用的，特别是还包括赋值操作。在进行详细阐述之前，首先要说明，声明类型和当前最确切类型的概念显然也可以方便地扩展到任意的标量表达式上，而不仅仅是针对标量变量。设 *X*是这样一个表达式，则：

- *X*有一个声明类型 *DT*(*X*) —— 更确切地说，是 *X*的计算结果拥有这样一个类型。显然 *DT*(*X*)是由表达式*X*中各个操作的声明类型而来（包括对 *X*中任一个操作调用的结果的声明类型）。*DT*(*X*)是在编译的时候确定的。
- *X*同样还有一个最确切类型 *MST*(*X*) —— 更确切地说，也是 *X*的计算结果拥有这样一个类型。*MST*(*X*)也明显地是来自于表达式 *X*中操作的当前值（包括对 *X*中任一个操作调用的结果的当前值），（通常）*MST*(*X*)直到运行的时候才会确定。

现在我们可以恰当地解释赋值了。考虑赋值操作

```
V := X ;
```

（其中*V*是一个标量变量，*X*是一个标量表达式）。*DT*(*X*)必须是*DT*(*V*)的子类型，否则赋值操作是不合法的（这是编译时进行的检查）。如果赋值操作是合法的，其结果是使得 *MST*(*V*)等于*MST*(*X*)、*v*(*V*)等于*v*(*X*)。

顺便提一句，如果变量 *V*的当前最确切类型是 *T*，则*T*的每个真超类型也是变量 *V*的“当前类型”。比如，如果变量 E（声明类型为 ELLIPSE）当前值的最确切类型是 CIRCLE，则CIRCLE、ELLIPSE和PLANE_FIGURE都是E的“当前类型”。但是，至少是非正式地，“*X*的当前类型”通常特指*MST*(*X*)。

2. 对可置换性的再讨论

考虑如下的操作定义：

```
OPERATOR COPY ( E ELLIPSE ) RETURNS ( ELLIPSE ) ;
RETURN ( E ) ;
```

```
END OPERATOR ;
```

根据可置换性，COPY操作的执行参数的最确切类型可以是 ELLIPSE或CIRCLE —— 而且无论是哪一种类型，其返回值显然也会具有同样的最确切类型。由此可见，可置换性的概念具有进一步的含义，即（一般而言），如果定义操作*Op*的返回值的声明类型为 *T*，则执行操作*Op*得到的实际结果可以属于 *T*的任意子类型。换句话说，就是 (a)（一般地）引用一个声明类型为*T*的变量，实际上得到的可能是 *T*的任意子类型的一个值；所以 (b)（同样，一般地）执行一个声明类型为*T*的操作，实际上返回的值可能属于 *T*的任何一个子类型。

3. 类型下移

同样用上面的例子：

```
VAR E ELLIPSE ;
VAR C CIRCLE ;
C := CIRCLE ( LENGTH ( 3.0 ), POINT ( 0.0, 0.0 ) ) ;
E := C ;
```

现在*MST*(E)为CIRCLE。假设要得到例子中圆的半径，并把它赋值给变量 L。我们也许会这样做：

```
VAR L LENGTH ;
L := THE_R ( E ) ;              /*编译时会出现类型错误！！！*/
```

但是，就像注释所指出的，上面的代码在编译时会出现类型错误。更确切地说，是由于赋值表达式右边的操作THE_R（取半径）需要一个类型为 CIRCLE的参数，而参数E的声明类型是ELLIPSE而不是CIRCLE。注意：如果编译时不进行类型检查，则假如运行的时候 E的当前值是一个椭圆而不是圆，我们会得到一个运行时的类型错误，这种情况就更糟糕了。当然，对于目前的情况，我们其实知道运行时 E的值是一个圆，问题是我们知道但是编译器不知道。

为了解决这一类的问题，引入一个新的操作，并非正式地以 TREAT DOWN（类型下移）来称呼它。则例子中获取半径的正确方法如下：

```
L := THE_R ( TREAT_DOWN_AS_CIRCLE ( E ) ) ;
```

定义表达式TREAT_DOWN_AS_CIRCLE(E)的声明类型为CIRCLE，这样编译时的类型检查就通过了。那么在运行的时候：

- 如果E的当前值确实是一个圆，则整个表达式正确地返回该圆的半径。确切地说，执行 TREAT DOWN会产生一个结果，比如说是Z，则（a）Z的声明类型*DT*(Z)等于CIRCLE，因为有形如“ ..._AS_CIRCLE ”的说明；（b）Z的当前最确切类型*MST*(*Z*)等于*MST*(E)，在这个例子中同样是 CIRCLE；（c）Z的当前值 *v*(Z)等于 *v*(E)；（d）表达式“ THE_R(Z) ”经过计算给出所要的半径（之后被赋值给 L）。
- 但是如果E的当前值只是属于类型 ELLIPSE而不是CIRCLE，则TREAT DOWN在运行时会出现类型错误。

通常，使用类型下移的目的在于保证如果出现运行时的类型错误，那么可以肯定是在调用TREAT DOWN的时候出现了错误。

注意：假设CIRCLE有一个真子类型，比如说是 O_CIRCLE（“ O-circle ”是指一个圆心在原点的圆）：

```
TYPE O_CIRCLE   POSSREP ( R LENGTH )
     SUBTYPE_OF (CIRCLE )
```

```
CONSTRAINT (THE_CTR ( O_CIRCLE ) = POINT ( 0.0, 0.0 ) ) ;
```

则变量E的最确切类型在某些时候就可能是O_CIRCLE而不是CIRCLE。如果是这样，则下面的TREAT DOWN操作

```
TREAT_DOWN_AS_CIRCLE ( E )
```

就可以成功执行，并生成结果，比如说是Z，并且（a）由于有“..._AS_CIRCLE”的说明，*DT*(Z)等于CIRCLE；（b）*MST*(Z)等于O_CIRCLE，因为E的最确切类型是O_CIRCLE；（c）*v*(Z)等于*v*(E)。换句话说，TREAT DOWN总会产生最确切类型，而不可能“向上提升”类型的层次。

为了便于今后的讨论，对于操作调用TREAT_DOWN_AS_*T*(X)，这里给出一个更加规范的语义说明，其中*X*是一个标量表达式。首先，最重要的是*T*必须是*DT*(*X*)的子类型（这是编译时进行的类型检查）。其次，*MST*(*X*)必须是*T*的子类型（这是运行时进行的类型检查）。如果这些条件都得到了满足，调用返回一个结果*Z*，并且*DT*(*Z*)等于*T*，*MST*(*Z*)等于*MST*(*X*)以及*v*(*Z*)等于*v*(*X*)。注意：文献[3.3]也定义了一个一般形式的TREAT DOWN，允许将一个操作数的类型“下移为（be treated down）”另外的类型，而不是某个明确命名的类型。

19.5 约束特化

考虑调用如下一个类型为ELLIPSE的选择子操作：

```
ELLIPSE ( LENGTH ( 5.0 ), LENGTH ( 5.0 ), POINT ( ... ) )
```

这个表达式返回一个半轴相等的椭圆。但是，在现实世界中半轴相等的椭圆实际上是圆，那么这个表达式能否返回一个最确切类型是CIRCLE而不是ELLIPSE的值呢？

对于类似这一类的问题在学术界曾经（事实上现在也仍然）引起非常多的争论。经过仔细考虑，我们决定在自己的模型中，最好还是让这样的表达式确实能够返回一个最确切类型为CIRCLE的值。更一般地，如果类型*T′*是类型*T*的子类型，而调用类型*T*的一个选择子操作会返回一个满足*T′*约束的值，则（在我们的模型里）选择子操作的调用结果就是一个类型为*T′*的值[⊖]。注意：在当今的商业系统中几乎没有哪个在实现的时候这样做，但是我们认为这正是那些系统的失败之处。文献[3.3]表明，由于这种缺陷，这些商业系统被迫支持“不是圆形的圆”、“不是正方形的正方形”以及类似的一些毫无意义的东西——但是我们的方法不会出现这样的后果。

由上述可知，（至少在我们的模型里）不存在最确切类型为ELLIPSE且*a*=*b*的值。换句话说，最确切类型为ELLIPSE的值确切地对应于现实世界中的椭圆而不会是圆。与此相反，在其他继承模型中，最确切类型为ELLIPSE的值对应于现实世界中的椭圆，但是这些椭圆可能是圆也可能不是圆。由此我们觉得我们的模型做为“现实的一个模型”更容易为人接受。

像*a*=*b*的椭圆必然属于CIRCLE类型这样的概念在文献[3.3]中被称为约束特化（specialization by constraint），但是必须说明的是，其他作者在使用这个词的时候可能有着完全不同的含义（例如，见文献[19.7]和[19.11]）。

1. 对THE_伪变量的再次讨论

⊖ 文献[3.3]建议通过类型T中的SPECIALIZE子句来达到这一效果，但是我们后来得出结论，要达到我们所需的效果并不需要特殊的语法。

在第5章中曾说过，THE_伪变量的作用是修改变量的某个分量而同时保持其他分量不变（这里的分量是指某种可能表示形式的分量，而不是实际表示形式的分量）。比如，变量E的声明类型是ELLIPSE，其当前值是一个a=5、b=3的椭圆。则赋值表达式：

```
THE_B ( E ) := LENGTH ( 4.0 ) ;
```

将把变量E的半轴b改为4，而保持半轴a和中心不变。

那么，如第8章的8.2节所说，THE_伪变量在逻辑上是不需要的——它们只是一种快捷的形式而已。比如，上面使用THE_伪变量的赋值表达式实际上可以用下面的表达式来代替，只是不够简洁[㊀]：

```
E := ELLIPSE ( THE_A ( E ), LENGTH ( 4.0 ), THE_CTR ( E ) ) ;
```

所以考虑如下的赋值：

```
THE_B ( E ) := LENGTH ( 5.0 ) ;
```

根据定义，这个表达式与下面的表达式等价：

```
E := ELLIPSE ( THE_A ( E ), LENGTH ( 5.0 ), THE_CTR ( E ) ) ;
```

约束特化在这时就起作用了（因为表达式的右边返回了一个a=b的椭圆），则最终效果是赋值以后*MST*(E)为CIRCLE而不是ELLIPSE。

接下来再看看下面的赋值操作：

```
THE_B ( E ) := LENGTH ( 4.0 ) ;
```

现在E包含一个a=5、b=4的椭圆（像前面一样），则*MST*(E)再次变为ELLIPSE——这是我们所说的约束泛化（generalization by constraint）的效果。

注意：假设（像我们在19.4节快结束时所做的那样）CIRCLE类型有一个真子类型O_CIRCLE（“O-circle”代表圆心在原点的圆）：

```
TYPE O_CIRCLE POSSREP ( R LENGTH )
SUBTYPE_OF ( CIRCLE )
CONSTRAINT ( THE_CTR ( O_CIRCLE ) = POINT ( 0.0, 0.0 ) ) ;
```

则变量E的当前值的最确切类型在某些时候就可能是O_CIRCLE而不是CIRCLE。如果是这样，那么考虑下面一系列的赋值操作[㊁]：

```
THE_A ( E ) := LENGTH ( 7.0 ) ;
THE_B ( E ) := LENGTH ( 7.0 ) ;
```

执行了第一个赋值操作之后，E将包含一个“真正的椭圆”，这是约束泛化的功效。在执行了第二个赋值操作之后，它又将变回一个圆——但是将变回一个确切的O_circle呢还是“仅仅是一个圆”呢？显然，我们希望它是一个O_circle。而事实上也的确是这样的，准确地说，是因为它满足类型O_CIRCLE的约束（包括从CIRCLE类型继承来的约束）。

2. 类型转化的分支问题

仍然让变量E的声明类型为ELLIPSE。我们已经看到如何将E的类型“下移”（比如，当其当前最确切类型是ELLIPSE时，如何将其当前最确切类型修改为CIRCLE）；也看到了如何将E的类型“上移”（比如，当其当前最确切类型是CIRCLE时，如何将其修改为ELLIPSE）。但

㊀ 顺便说明一下，TREAT DOWN(类型下移)也可以被用做伪变量[3.3]，但这样做同样也是为了简便。

㊁ 在第8章曾经提到，文献[3.3]提供了一种多重赋值形式，可以使这一系列的赋值执行起来像一个赋值操作一样。

是如何处理类型转化的“分支”问题呢？假设我们对例子中的类型 ELLIPSE进行扩展，使其有CIRCLE和NONCIRCLE（非圆）（含义显而易见）两个直接子类型 ㊀。无需进行过多细节的讨论，很清楚会有如下的结果：

- 如果E的当前值属于类型 CIRCLE（即 $a=b$），则修改 E使得 $a>b$会让 $MST(E)$变成NONCIRCLE；
- 如果E的当前值属于类型 NONCIRCLE（即 $a>b$），则修改E使得 $a=b$会让 $MST(E)$变成CIRCLE；

这样，约束特化同样可以解决类型改变的分支问题。注意：如果想知道的话，实际上修改 E使得 $a<b$是不可以的（这与类型ELLIPSE的约束冲突）。

19.6 比较

两个变量E和C的声明类型仍然是ELLIPSE和CIRCLE，假设用C的当前值给E赋值：

```
E := C ;
```

显然，如果现在进行相等性比较

```
E = C
```

得到的结果应该为“真”——事实也确实如此。而一般的规则如下。考虑形如 $X=Y$的比较（X和Y是标量表达式）。声明类型 $DT(X)$必须是声明类型 $DT(Y)$的子类型或类似的情况，否则比较是不合法的（这是编译时进行的类型检查）。如果比较是合法的，并且如果最确切类型 $MST(X)$等于最确切类型 $MST(Y)$且值 $v(X)$等于值 $v(Y)$，则其返回结果是“真”，否则为“假”。尤其要注意的是如果两个值的最确切类型不同，则它们不能进行“相等性比较”。

1. 对关系代数的影响

在关系代数的许多操作中总会显式或隐式地牵扯到相等性比较，当涉及到超类型和子类型的时候，其中某些操作所表现出来的状态可能会让人觉得与想像中有些不一样（至少咋一看是有些不一样）。考虑如图19-2所示的两个关系RX和RY，可以看到RX唯一的属性A的声明类型是ELLIPSE，而相应地，在RY中A的声明类型为CIRCLE。如图中所示，我们用形如 Ei的标识符来表示不是圆的椭圆，用 Ci来表示圆。最确切类型用斜体表示。

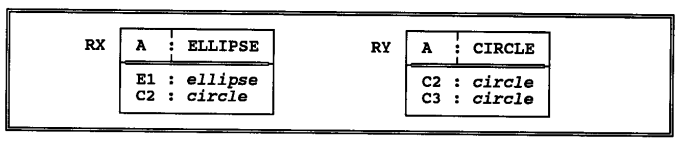

图19-2 关系RX和RY

现在考虑RX和RY的连接RJ（见图19-3）。显然RJ中的每个属性 A的值都必须属于类型CIRCLE（因为对于RX中任何属性A的值，其最确切类型如果是 ELLIPSE的话，是不能和RY中属性A的值进行“相等性比较”的）。由此也许会认为RJ中属性A的声明类型应该是CIRCLE而不是ELLIPSE。但是先让我们考虑一下下面的问题：

- 既然RX和RY都只有一个属性A，则RX JOIN RY可以简化成RX INTERSECT RY。那么在这样的情况下，确定 JOIN的结果中属性的声明类型的规则显然也需要简化成针对

㊀ 顺便说一句，ELLIPSE现在成了一个哑类型（见19.7节）。

INTERSECT的类似规则。

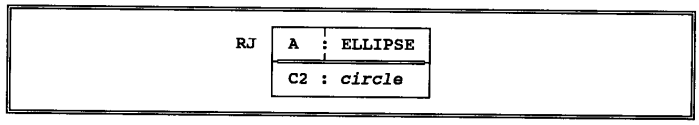

图19-3 关系RX和RY的连接RJ

- RX INTERSECT RY逻辑上等于RX MINUS (RX MINUS RY)。设第二个操作——即RX MINUS RY——的结果为RZ。则显然有：
  a. 一般地讲，某些RZ中属性A的值的最确切类型为ELLIPSE，所以RZ中属性A的声明类型一定是ELLIPSE。
  b. 原来的表达式由此就变成了RX MINUS RZ，而在RX和RZ中，属性A的声明类型都是ELLIPSE。因此得到的最终结果中，属性A的声明类型显然又一定成了ELLIPSE。
- 由此可见，INTERSECT操作的结果中，其属性的声明类型一定是ELLIPSE而不是CIRCLE，因此JOIN的结果也是一样——即使是该属性中的每个值的声明类型事实上都必须是CIRCLE！

现在来看看关系的差操作符——MINUS。首先考虑一下RX MINUS RY。很明显，该操作的结果属性A的某些值是ELLIPSE而不是CIRCLE，所以结果中属性A的声明类型也一定是ELLIPSE。那么RY MINUS RX又怎么样呢？显然，这个操作的结果中A的每个值都属于类型CIRCLE，所以很自然地会想到，结果中A的值的声明类型会是圆而不是椭圆。但是可以发现，RX INTERSECT RY在逻辑上不仅与RX MINUS (RX MINUS RY)等价，就像刚才讨论过的那样，而且与RY MINUS (RY MINUS RX)等价。由此很显然在RY MINUS RX的结果中，指定A的声明类型为CIRCLE会引起矛盾。可见，即使是在RY MINUS RX的情况下，每个属性值的类型实际上是CIRCLE，MINUS操作的结果属性的声明类型也必须为ELLIPSE而不是CIRCLE。

最后来考虑一下RX UNION RY。通常在这种情况下，结果属性A的某些值的最确切类型显然会是ELLIPSE，则结果属性A的声明类型也必须是ELLIPSE。这样，UNION结果属性的声明类型也必须为ELLIPSE（但是就这里的情况而言，与JOIN、INTERSECT和MINUS不同，UNION的结果在感觉上并不违反常规）。

下面是一般的规则：

- 设*rx*与*ry*是具有公共属性A的关系，并且用*DT*(*Ax*)和*DT*(*Ay*)相应地代表*A*的声明类型。考虑*rx*与*ry*的连接（当然是在属性*A*上或至少是包括属性*A*）。*DT*(*Ax*)必定是*DT*(*Ay*)的子类型，或者反之，否则连接是不合法的（这是编译时进行的类型检查）。如果连接是合法的，不失一般性，设*DT*(*Ay*)是*DT*(*Ax*)的子类型，则结果中属性A的声明类型为*DT*(*Ax*)。
- 类似的分析也适用于“并、交、差”操作。在每种情况下，（a）操作数的相应属性的声明类型中，一种应该是另一种的子类型；（b）结果中相应属性的声明类型是两个类型中相对不确切的那一个（所谓两个类型*T*和*T*′中有一个相对不确切的含义，是指一个是另一个的子类型，意思是任何一个都可以是超类型）。

2. 类型检测

在前面的一节中，我们给出了一段带有形如IS_SQUARE、IS_CIRCLE等操作的代码，用

来检测一个指定的值是否具有特定的类型。现在可以来进一步探讨这些操作。首先假设定义一个类型*T*会自动定义一个如下形式的判真（truth-valued）操作

```
IS_T ( X )
```

其中*X*是一个标量表达式，且*DT*(*X*)是*T*的超类型（这是编译时进行的类型检查）。如果*X*属于类型*T*，则返值为真，否则为假。再次强调参数*X*的声明类型必须是类型*T*的超类型。例如，如果C是一个CIRCLE类型的变量，则表达式

```
IS_SQUARE ( C )
```

是不合法的（在编译的时候会出现类型错误）。另一方面，下面的两个表达式都是合法的，并且返回值都是真：

```
IS_CIRCLE ( C )
IS_ELLIPSE ( C )
```

如果E是类型为ELLIPSE的变量，但是当前的最确切类型是 CIRCLE的某个子类型，则表达式

```
IS_CIRCLE ( E )
```

的返回值同样是真[⊖]。

还假设定义一个类型*T*会自动定义一个如下形式的操作：

```
IS_MS_T ( X )
```

其中*X*是一个标量表达式，且*DT*(*X*)是*T*的超类型（这同样也是编译时进行的类型检查）。如果*X*的最确切类型属于*T*，则返回值为真，否则为假。注意：这样的说法也许有助于理解，比如用自然语言，操作IS_ELLIPSE最好翻译成“是一个椭圆”，而操作IS_MS_ELLIPSE翻译成“确实是一个椭圆”更合适。

下面是一个关于矩形、正方形和IS_MS_RECTANGLE的例子：

```
VAR R RECTANGEL ;
IF IS_MS_RECTANGLE ( R )
   THEN CALL ROTATE ( R );
END IF ;
```

从例子中可以直接看到，（a）ROTATE是一个用于让参数矩形绕其中心旋转90°的操作；（b）如果矩形碰巧是一个正方形，则旋转就毫无意义。

类型检测还与关系操作有关。看看下面这个例子。关系变量 R有一个类型为ELLIPSE的属性A。假设我们希望获得R中所有A的值为圆、且半径大于2的元组，则我们也许会这样做：

```
R WHERE THE_R ( A ) > LENGTH ( 2.0 )
```

但是这个表达式在编译的时候会出现类型错误，因为 THE_R需要一个类型为CIRCLE的参数，但是A的类型为ELLIPSE而不是CIRCLE（当然，如果不在编译时做类型检查，那么假如在运行时遇到一个元组，其中A的值为椭圆而不是圆的话，就会出现运行时的类型错误。）

显然，我们需要做的就是，在检查半径之前就把 A的值是椭圆的元组过滤掉。下面的表达式完成的就是这一功能：

```
R : IS_CIRCLE ( A ) WHERE THE_R ( A ) > LENGTH ( 2.0 )
```

⊖ 文献[3.3]中定义了本小节中所有“类型检查”算符的通用形式——比如一个通用形式的IS_T用来检测一个操作数是否与另一个操作数有同样的类型，而不仅仅是检测其是否属于某个明确命名的类型。

简单地说，这个表达式是用来返回A的值为半径大于2的圆的元组。更准确一些，它返回了这样一个关系

a. 属性名称与R一样，只是结果中属性A的类型是CIRCLE而不是ELLIPSE；

b. 关系中只包含来自于R、且A的值的类型是CIRCLE、且半径大于2的元组。

换句话说，我们所讨论的是一个新的关系操作，其形式如下

```
R : IS_T ( A )
```

其中*R*是关系表达式，*A*是该表达式所指关系——比如说是*r*——的一个属性。*A*的声明类型*DT*(*A*)必须是*T*的一个超类型（这是编译时进行的类型检查）。表达式的所有值定义为一个关系：

a. 属性名称与*r*一样，除了属性*A*的类型在*r*中为*T*；

b. 结果集是由来自关系*r*、且属性*A*的值属于类型*T*的元组构成，只是在新关系中这些元组中属性*A*的声明类型为*T*。

我们还可以定义一个如下形式的新关系操作

```
R : IS_MS_T ( A )
```

其处理方式类似。

19.7 操作、版本和签名

在19.3节中曾说过，一个给定的操作在相同的接口形式下可以有多个不同的实现版本（也叫做显式特化（explicit specialization））。也就是说，当顺着类型层次结构从超类型*T*下溯到子类型*T'*时，（由于各种原因）我们必须能够针对*T'*重实现类型*T*的操作。举个例子，考虑下面的MOVE操作：

```
OPERATOR MOVE ( E ELLIPSE, R RECTANGLE ) RETURNS ( ELLIPSE )
VERSION ER_MOVE ;
RETURN ( ELLIPSE ( THE_A ( E ), THE_B ( E ), R_CTR ( R ) ) ;
END OPERATOR ;
```

简单地说，操作MOVE的作用是“移动”一个椭圆，使其中心与矩形R的中心重合。或者说得更明确一些，它返回一个与参数椭圆E一样的椭圆，只是这个椭圆的中心与参数矩形R的中心重合。注意第二行的VERSION子句，它为这个特定版本的MOVE（见下一段）引入了一个不同的名字——ER_MOVE。还要注意的是假设存在操作R_CTR，其功能是返回指定矩形的中心点。

现在假设MOVE的一个显式特化，即另一个版本的定义是移动圆而不是椭圆[⊖]：

```
OPERATOR MOVE ( C CIRCLE, R RECTANGLE ) RETURNS ( CIRCLE )
VERSION CR_MOVE ;
RETURN (CIRCLE ( THE_R ( C ), R_CTR ( R ) ) ;
END OPERRATOR;
```

通过类似的方法，还可以得到其他情况下的显式特化，比如参数的最确切类型分别是ELLIPSE和SQUARE，相应地可以有ES_MOVE；以及参数的最确切类型分别是CIRCLE和SQUARE，相应地可以有CS_MOVE。

⊖ 实际上在这个例子中定义这样一个显式实现的意义并不大（那又是因为什么呢？）。

1. 签名

简而言之，签名就是某个操作的名字和该操作的操作数类型的结合体[⊖]。（需要顺便提一下，不同的作者和不同的语言对这个词会有一些稍微不同的解释。比如，结果类型有时候会被认为是签名的一部分，而操作数和结果的名字有时候也是这样）。但是，我们还是要认真地再回顾一下：

a. 参数和参变量之间的区别；

b. 声明类型和实际（最确切）类型之间的区别；还有

c. 用户所见的操作和系统所见的操作（是指在接口形式下那些操作的显式特化或版本，就像上面所提到的）之间的区别。

实际上，虽然文字上经常区分不开！但是，对于一个给定的操作 *Op*，我们至少可以区分出三种不同的签名：

- 单个的描述签名（specification signature），它由两部分构成，一部分是操作的名字 *Op*，另一部分是*Op*的参变量的声明类型，它们按照定义 *Op*时提供给用户的顺序排列。这个签名对应于用户所接受的操作 *Op*。比如MOVE的具体签名就是 MOVE (ELLIPSE, RECTANGLE)[⊜]。
- 一组版本签名（version signature）。对应于*Op*的每个显式特化或实现版本都有一个版本签名，它由操作*Op*的名称和该版本参变量的声明类型构成，声明类型是按照定义的顺序排列的。这些签名对应于 *Op*实现代码的不同部分。比如，对于 MOVE来说，版本CR_MOVE的版本签名是MOVE(CIRCLE, RECTANGLE)。
- 一组调用签名（invocation signature）。操作的每个参数都可以有若干个最确切类型，各个参数的最确切类型的所有可能的组合都对应一个版本签名，它由操作 *Op*的名称加上参数最确切类型的某种组合情况一起构成。类型是按照定义的顺序排列的。这些签名对应于*Op*的各种可能的调用情况（当然这种对应是一对多的 —— 即一个调用签名可以对应多个不同的调用）。比如，设E和R的最确切类型分别是CIRCLE和SQUARE，则对于MOVE的调用MOVE(E,R)来说，调用签名是MOVE(CIRCLE, SQUARE)。

因此，同一个操作的不同调用签名至少是潜在地对应着该操作的不同执行版本（即对应着相同接口下的不同特化）。这样，如果“同一个”操作的接口下确实存在着多个版本，则在特定情况下要调用哪一个版本取决于哪一个的版本签名和应用的调用签名“最匹配”。决定是否最匹配的过程 —— 即决定调用哪一个版本的过程 —— 当然是一个在19.3节讨论过的动态联编的过程。

顺便要提起注意的是，（a）描述签名真正是一个属于模型的概念；（b）版本签名仅仅是一个属于实现的概念；（c）调用签名虽然从某个角度上说是一个属于模型的概念，但是和可

⊖ 它等同于标识 —— 译者注

⊜ 文献[3.3]建议可以将一个给定操作的具体签名的定义和其所有实现（版本）的定义区分开来。最基本的想法是要支持哑(dummy)类型(也叫做“抽象的”或“非实例化的”类型，或有时只是叫做“接口”)—— 即不是任何值的最确切类型的类型。这种类型为引入可以适用于不同的正规类型的操作提供了一种方法，这些不同的类型都是该哑类型的真正子类型。这样的一个操作接着就可以针对那些常规的子类型进行显式的专门处理 —— 即明确定义该操作的一个合适的版本。就我们的例子而言，PLANE_FIGURE就是一个所谓的哑类型；操作AREA的描述签名可以在PLANE_FIGURE层进行定义，而ELLIPSE、POLYGON等类型的显式实现代码(版本)会在接下来定义。

置换性一样，它实际上首先是类型继承的一个直接的逻辑推论。有可能存在不同的调用签名这一事实其实只是可置换性概念的一部分。

2. 只读操作与更新操作

迄今为止，我们一直默认MOVE是一个只读操作，但是假设现在把它作为一个修改操作：

```
OPERATOR MOVE ( E ELLIPSE, R RECTANGLE ) UPDATES ( E )
VERSION ER_MOVE ;
BEGIN
   THE_CTR ( E ) := R_CTR ( R ) ;
   RETURN ;
END ;
END OPERATOR ;
```

（提醒一下，只读操作和修改操作有时候分别被叫做观察子（ observer）和变异子(mutator)。如果你需要搞清楚两者之间的差异，请参阅第5章。）

现在可以看到，调用这个版本的 MOVE会修改它的第一个参数（简单地说是改变了该参数的中心）。进一步还可以发现，无论第一个参数的最确切类型是 ELLIPSE还是CIRCLE，修改都会起作用，换句话说，对于圆，显式特化是不需要的 ㊀。因此，一般而言，修改操作的好处之一就在于它可以不必明确地编写某些操作特化。尤其是对程序维护具有特别的意义，比如，如果引入O_CIRCLE做为CIRCLE的子类型会出现什么情况呢？

3. 改变操作的语义

当我们顺着类型层次结构下溯时，对于操作的重实现至少是合法的，这一事实有一个非常重要的后果：它使得改变操作的语义成为可能。比如，就AREA而言，可能会出现这种情况，类型CIRCLE的实现版本实际上返回的是圆周长而不是圆面积（细致的类型设计可以有助于在某种程度上缓和这一问题，比如，如果操作 AREA的返回类型被定义为AREA类型，显然执行是不可能返回一个LENGTH类型的结果，但是它仍然可能返回一个错误的面积！）。

然而（虽然看起来有些让人吃惊），甚至会有人说 —— 实际上已经这样提出了 —— 这种方式的语义改变是值得的。比如，设类型 TOLL_HIGHWAY是类型HIGHWAY的真子类型，而操作TRAVEL_TIME的作用是计算在指定的高速公路上通过指定两点之间的行进时间。对于要征收过路费的高速公路来说，计算公式是 $(d/s) + (n*t)$，其中d=距离，s=速度，n=收费站的数目、t=在收费站停留的时间。对于不收费的高速公路，计算公式则相应地为 d/s。

再举一个反例，即说明确实不欢迎语义改变的情况的例子，再次考虑椭圆和圆。应该说我们希望操作AREA的定义对于同一个圆应该能够返回同一个面积值，无论它是一个圆还是一个椭圆。换言之，假设会顺序发生如下的事情：

1) 定义类型ELLIPSE及相应版本的AREA操作。为简单起见，设在 AREA的代码中并没有使用椭圆的实际表示形式。
2) 定义类型CIRCLE为类型ELLIPSE的子类型，但是还没有针对圆定义一个 AREA操作的单独实现版本。
3) 对于某个特定的圆调用 AREA，得到结果area1，调用的当然是ELLIPSE版本的AREA。
4) 现在针对圆定义一个AREA操作的单独实现版本。
5) 对于同一个圆再次调用AREA，得到结果area2（而这一次调用的是CIRCLE版本的AREA）。

㊀ 事实上，如果约束特化得到支持的话，它们就完全不需要了。

之后我们肯定要求“应该有”area1 = area2，但是这一“应该有”的要求并不是强制的。即像已经说过的那样，总有可能针对圆的AREA的实现版本会返回一个周长而不是面积，或是返回一个错误的面积。

让我们再回到TRAVEL_TIME的例子。事实上，我们发现这个例子以及类似的例子是极其让人无法信服的——即无法让人相信在例子所展现的情况下，改变一个操作的语义可以被认为是有必要的。考虑下面内容：

- 如果TOLL_HIGHWAY确实是HIGHWAY的一个子类型，这意味着根据定义，每一条独立的收费高速公路首先是一条高速公路。
- 因此，某些高速公路（即某些HIGHWAY类型的值）实际上是收费高速公路——路上有收费站。所以HIGHWAY类型并不是“没有收费站的高速公路”，而是“可以有*n*个收费站的高速公路”（*n*可以等于零）。
- 故而类型HIGHWAY的操作TRAVEL_TIME并不是“计算在一条没有收费站的高速公路上的行进时间”，而是“计算忽略了收费站后在一条高速公路上的行进时间 *d*/*s*”。
- 相应地，类型TOLL_HIGHWAY的操作TRAVEL_TIME是“在不忽略收费站的情况下，计算在一条高速公路上的行进时间 (*d*/*s*) + (*n***t*)”。所以实际上这两个TRAVEL_TIME在逻辑上是不同的操作。由于这两个不同的操作具有相同的名字，这样就会产生混淆。实际上在这里我们引入了“重载多态”而不是“包含多态”。

（作为说明，我们指出在实践中还会有进一步的混淆，因为非常遗憾的是在谈到包含多态的时候许多作者实际使用的却总是“重载”这个词。）

总而言之，改变操作的语义并不是一个好的想法。像我们已经看到的那样，这种要求不是强制的。但是我们当然可以定义自己的继承模型——我们也确实这样做了，在模型中如果出现语义被改变的情况，则执行是不合法的（即该执行是不属于模型的，其含义是不可预知的）。应该注意到，我们对于这一问题的立场（即这种改变是不合法的）确实是有好处的，无论给定操作*Op*是否定义了任何显式特化，用户的感觉都是一样的：也就是（a）只存在一个操作，一个唯一的操作叫做*Op*；（b）该操作对于属于某个特定类型*T*的参数值有效，因此，根据定义对于属于*T*的真子类型的参数值也有效。

19.8 一个圆是一个椭圆吗

圆真的是椭圆吗？到现在为止我们在本章中都假设——而且有充分的理由假设——是这样的。但是现在我们必须面对这样一个事实，即这件非常显而易见的事情在学术界有很多争论。考虑我们常用的变量E和C，其声明类型分别为ELLIPSE和CIRCLE。假设这些变量做了如下的初始化：

```
E := ELLIPSE ( LENGTH ( 5.0 ), LENGTH ( 3.0 ),
                              POINT ( 0.0, 0.0 ) ) ;
C := CIRCLE  ( LENGTH ( 5.0 ), POINT ( 0.0, 0.0 ) ) ;
```

特别要注意到的是THE_A(C)和THE_B(C)的值现在都是5。

现在我们肯定可以对E做的一个操作是“修改半轴*a*”，比如：

```
THE_A ( E ) := LENGTH (6.0 ) ;
```

但是如果我们要对C执行类似的操作

```
THE_A ( C ) := LENGTH (6.0 ) ;
```

就会出错！到底是什么错误呢？让我们来看看，如果确实进行了修改，则变量 C将不再包含一个“圆”，这与对于圆的约束 $a=b$相冲突（a现在是6，而b应该还是5，因为我们还没有对其进行改变）。换句话说，C现在将包含一个“非圆形的圆”，从而与类型 CIRCLE的类型约束冲突。

既然“非圆形的圆”与逻辑和通常的感觉相冲突，那么首先不允许进行这种修改看起来是合理的。明显的方法是在编译的时候拒绝这种操作，即进行这样的定义，对于半轴 a或b的修改说赋值是不合语法的。换句话说，对于 THE_A和THE_B的赋值不适用于类型 CIRCLE，而进行这种修改的尝试会因为编译时的类型错误而失败。

注意：实际上这种赋值显然是不合语法的，我们曾说过，对 THE_伪变量的赋值实际上是一种缩写。比如对THE_A(C)的赋值，如果是合法的话，将是类似如下形式的表达式的缩写：

```
C := CIRCLE ( ... ) ;
```

于是表达式右边的 CIRCLE选择子操作调用会包括一个值为 LENGTH(6.0)的THE_A参数，但是CIRCLE选择子操作需要的不是 THE_A参数，而是THE_R参数和THE_CTR参数。所以原来的赋值显然是不合法的。

1. 改变语义会发生什么

为了维护这样的概念，即对于圆而言，对 THE_A和THE_B的赋值还是合法的，常常会提出如下的想法，即如果参数是一个圆，对于如 THE_A的赋值应该重新定义 —— 换句话说，要显式特化 —— 也就是在这种情况下也要同时对 THE_B赋值，这样，这个圆在修改之后仍然满足$a=b$的约束。但是这种想法是我们要立即阻止的。之所以反对这一想法，至少是基于以下三个原因：

- 首先，对THE_A和THE_B赋值的语义在继承模型里是 —— 非常慎重地 —— 规定的，不可以用上面所说的方式进行改变。
- 其次，即使模型没有规定这些语义，但是我们已经说明（ a）通常随意改变一个操作的语义是一个不好的想法，（b）而这样去改变一个操作的语义还会产生副作用，就更不好了。保持一个操作的效果能够正好满足要求，不多也不少，这是一个很好的普遍性原则。
- 第三，也是最重要的。退一步讲，能够像上面所说的那样进行语义改变的机会也并不是总能有的。比如，设类型 ELLIPSE有另外一个直接子类型 NONCIRCLE；同时对于“非圆形”有约束$a>b$；再假设如果可以的话，认为对于一个非圆形的 THE_A操作的赋值会使得$a=b$。那么对于这一赋值来说，其合适的语义重定义又是什么呢？准确地说，产生什么样的副作用是合适的呢？

2. 一个合理的模型确实存在吗

我们现在面临这样一种状况，即对 THE_A和THE_B的赋值操作普遍适用于椭圆却并不特定地适用于圆。但是：

a. 类型CIRCLE被假设为类型ELLIPSE的子类型；

b. 类型CIRCLE是类型ELLIPSE的子类型意味着，普遍适用于椭圆的操作也特定地适用于圆 —— 换句话说，操作是被继承的。

c. 但是我们现在却说对THE_A和THE_B的赋值操作没有被继承下来。

这样我们不是自相矛盾了吗？我们该怎么办呢？

在回答这些问题之前，需要先强调一下这一问题的严肃性。前面的讨论看起来真的像一个引线（threadpuller）。因为，如果类型CIRCLE没有从类型ELLIPSE继承这个操作，那我们又凭什么说一个圆“是”一个椭圆呢？如果某些操作事实上最终无法得到继承，那么“继承”又意味着什么呢？一个合理的继承模型确实存在吗？我们试图找到这样一个模型是不是一种幻想？

注意：一些作者确实提出——是郑重地提出——对THE_A的赋值对于圆和椭圆都应该有效（对于一个圆而言，它修改的是半径），而对THE_B的赋值只对于椭圆有效，所以事实上ELLIPSE应该是CIRCLE的子类型！换句话说，我们把类型的层次结构颠倒过来了。但是，只要稍微动动脑子，就足以发现这种想法是不可能实现的，特别是可置换性将会被破坏（一个一般意义下的椭圆，它的半径是什么？）。

恰恰是上面的那些想法使得某些作者得出结论，一个合理的继承模型是没有的（见本章结尾“参考文献和简介”中文献[19.1]的注释）。另一些作者则提出了一些继承模型，这些模型具有不合常理的或是不必要的特征与功能。比如，SQL3允许有“非圆形的圆”以及其他一些毫无意义的概念；事实上，如同SQL/92一样，它根本不支持类型约束。而允许出现那些毫无意义的概念的首要原因，就是对这种约束的省略（见附录B）。

3. 解决的办法

小结一下讨论到现在的情况，发现我们面对着以下的困境：

- 如果圆从椭圆那里继承了“向THE_A和THE_B赋值”的操作，则我们会得到非圆形的圆。
- 防止出现非圆形的圆的办法是支持类型约束。
- 但是如果支持类型约束，则上面的操作无法得到继承。
- 所以最后没有继承存在！

怎样解决这一困境呢？

出路在于——就像经常提起的那样——要认清这样一个事实（并按这一事实去处理问题），即值与变量之间在逻辑上存在着很明显的区别。在说“每个圆都是一个椭圆”的时候，更准确地说，我们的意思是如果一个值是圆，那它也是一个椭圆的值。我们当然不是说每一个圆的变量也是一个椭圆的变量（即一个声明类型为CIRCLE的变量不可以是一个声明类型为ELLIPSE的变量，而且也不能包含一个确切类型属于ELLIPSE的值）。换句话说，继承只适用于值，不适用于变量。比如，对于椭圆和圆来说：

- 像刚才说过的，一个值是圆，那它也是一个椭圆。
- 因此，所有适用于椭圆的、对值的操作也适用于圆。
- 但是我们对于任何值都不能做的一件事就是改变它！——如果可以改变它，它就不再是原来的那个值了（当然，我们可以通过修改一个变量来“改变它的当前值”，但是——重申一遍——不能以同样的方式改变一个值）。

现在，准确地说，适用于椭圆的值的操作是针对类型ELLIPSE定义的只读操作，而修改ELLIPSE变量的操作当然就是针对该类型定义的修改操作。因此，我们所声明的“继承只适用于值，而不适用于变量”，可以作如下更精确的表述：

- 只读操作是由值继承的，因此也毫无疑问可以由变量的当前值继承（当然是因为做为变

量当前值的那些值可以毫无妨碍地适用于只读操作)。

这个更为精确的表述也同时可以解释，为什么多态性和可置换性的概念是特别针对值而不是针对变量的。比如（只是为了提醒大家），无论系统在何处需要一个类型 *T* 的值，我们总可以用一个类型 *T* ′的值来置换，其中 *T* ′是 *T* 的子类型。实际上，我们是在引入值的可置换性原则的时候提到这一原则的。

那么对于修改操作又如何呢？由定义可知，这些操作适用于变量而不适用于值。那么我们是否可以说适用于ELLIPSE变量的修改操作会被CIRCLE变量继承呢？

不，我们不可以 —— 不完全可以。比如，对THE_CTR的赋值对于两种类型的变量都适用，但是（就像我们已经看到的），对于THE_A的赋值就不可以。这样，对于修改操作的继承是有条件的。事实上，必须明确地指出哪些修改操作是被继承的。比如：

- 类型ELLIPSE的变量具有修改操作 MOVE（修改版本）和对 THE_A、THE_B以及THE_CTR的赋值操作。
- 类型CIRCLE的变量具有修改操作MOVE（修改版本）和对THE_CTR以及THE_R而不是THE_A、THE_B的赋值操作。

注意：MOVE操作是在上一节讨论的。

当然，如果一个修改操作是通过继承得到的，我们就有了不仅仅适用于值而且适用于变量的多态性和可置换性。比如，修改版本的 MOVE需要一个类型为 ELLIPSE的变量做参数，但是在调用它的时候，也可以用一个类型为 CIRCLE的变量做参数。这样我们可以（而且确实是在）合理地探讨变量的可置换性原则 —— 但是这一原则比上面讨论的值的可置换性原则要有更多的限制。

19.9 约束特化 —— 再次讨论

对于前面几节的讨论，我们还需要加上一个很短但却是非常重要的后记。后记涉及类似这样的例子：“设CIRCLE类型有一个叫做 COLORED_CIRCLE（有颜色的圆）的真子类型”（意思是说“有颜色的圆”被认为是圆的一种特殊情况）。具有这种一般性质的例子在各种著作中经常被引用。只是我们还是要说这种例子并不是能令人信服的 —— 甚至在某些重要的方面，还会产生误导。说得更具体一些，在我们所探讨的情况中，认为有颜色的圆是圆的某种特殊情况实际上是毫无意义的。毕竟有颜色的圆肯定是作为一个图像（ image）来定义的，可能是在一个显示屏上，而普遍意义上的圆是几何图形而不是图像。因此认为COLORED_CIRCLE是一个完全单独的类型，而不是 CIRCLE的子类型也许看起来要更合理。这个单独的类型也许有这样一种可能的表示形式，它有一个分量的类型为CIRCLE，另一个的类型为COLOR，但是 —— 再说一遍 —— 它不是CIRCLE的子类型。

1. 继承可能的表示形式

有如下一个非常有力的理由支持上面的立场。首先，回到我们比较经常使用的关于椭圆和圆的例子。这里再次（部分地）给出它们的类型定义：

```
TYPE ELLIPSE POSSREP ( A LENGTH, B LENGTH, CTR POINT ) ... ;
TYPE CIRCLE POSSREP ( R LENGTH, CTR POINT ) ... ;
```

尤其要注意的是，椭圆和圆所声明的可能表示形式是不同的。但是，椭圆的可能表示形式 —— 必然，即使是隐含地 —— 也是圆的可能表示形式，因为圆是椭圆。即很自然地，圆通

过其*a*、*b*半轴（以及其中心）“来表示是可能的”，即使事实上其*a*、*b*半轴是一样的。当然，反之就不可以了——即圆的一种可能的表示形式未必是椭圆的一种表示形式。

由此我们可以认为，类似操作和约束这样的可能表示形式，是圆可以从椭圆继承到的进一步“属性”，或者更一般地认为是子类型可以从超类型继承到的进一步“属性”[⊖]。但是（当我们转向圆和有颜色的圆的情况时），很显然声明用于类型CIRCLE的可能表示形式并不能作为类型COLORED_CIRCLE的可能表示形式，因为其中没有可以表示颜色的部分！这一事实有力地说明了，有颜色的圆不是圆与圆是椭圆是具有同样的含义的。

2. 子类型到底意味着什么呢

下面要讨论的与前一个问题有（某些）联系，但是它的结论更强（即逻辑上更强）。结论是：通过约束特化无法从圆获得一个有颜色的圆。

为了解释这个结论，我们先回到椭圆和圆的情况。再次给出类型的定义：

```
TYPE ELLIPSE POSSREP ( A LENGTH, B LENGTH, CTR POINT ) ...
     CONSTRAINT ( THE_A ( ELLIPSE ) ≥ THE _B ( ELLIPSE ) ) ;
TYPE CIRCLE POSSREP ( R LENGTH, CTR POINT )
     SUBTYPE_OF ( ELLIPSE )
     CONSTRAINT ( THE_A ( CIRCLE ) = THE_B ( CIRCLE ) ) ;
```

就像我们以前看到的，类型CIRCLE的CONSTRAINT子句保证了一个满足*a*=*b*的椭圆会自动地被限定为CIRCLE类型。但是——现在回到圆和有颜色的圆的情况——对于类型COLORED_CIRCLE，我们无法给出任何类似的CONSTRAINT子句，来把一个圆限定为一个有颜色的圆。即我们无法给出任何类型约束，如果一个给定的圆满足这些约束，则意味着这个圆实际上是一个有颜色的圆。

因此，这再一次说明，认为COLORED_CIRCLE和CIRCLE是完全不同的类型是更为合理的；同时特别地认为类型COLORED_CIRCLE有这样一种可能的表示形式，即它的一个分量是CIRCLE类型的，而另一个是COLOR类型的：

```
TYPE COLORED_CIRCLE POSSREP ( CIR CIRCLE, COL COLOR ) ... ;
```

事实上，我们在这里接触到了一个更大的问题。事实是，我们相信子类型总是通过约束特化得到的！也就是说，我们认为如果*T′*是*T*的子类型，则总会有一个这样的类型约束，如果类型*T*的值满足了这个约束，则这个值实际上是类型*T′*的一个值（而且应该自动地被限定为类型*T′*）。假设*T*和*T′*表示两个类型，*T′*是*T*的子类型（实际上，可以不失一般性地假设*T′*是*T*的直接子类型），则：

- *T*和*T′*都是基本集合（是值的集合），同时*T′*是*T*的子集。
- *T*和*T′*都有成员谓词——即当且仅当满足这一谓词的时候，一个值才是上述集合的成员（因此也就是相应类型的成员）。设这些谓词分别为*P*和*P′*。
- 现在注意到，根据定义，谓词*P′*在某个值是属于类型*T*的值时取真值。这样，它实际上可以写成是针对属于*T*的值的公式（要好于针对*T′*的值）。

⊖ 在形式模型中我们并不这样认为——即不把这样继承得到的可能表示形式看作是被声明的表示形式——因为说它是被声明的表示形式会造成矛盾。具体而言，如果说类型CIRCLE从类型ELLIPSE继承了一种可能表示形式，则文献[3.3]会要求对于一个CIRCLE变量的THE_A、THE_B的赋值是合法的，但是我们知道这是不合法的。这样，说类型CIRCLE从类型ELLIPSE继承了一种可能表示形式只是一种facon de parler——没有任何正式的含义。

- 用针对*T*的值的公式表达的谓词*P'*恰好是一个类型约束，一个属于*T*的值要成为一个属于*T'*的值就必须满足它。换句话说，一个属于*T*的值会被准确地限定为属于类型*T'*，如果它满足谓词*P'*。

因此，约束特化是唯一在概念上定义子类型的有效方法。做为一个推论，我们拒绝接受类似这样的例子，即认为COLORED_CIRCLE是CIRCLE的一个子类型。

19.10 小结

我们已经简要地介绍了类型继承模型的基本概念。如果类型*B*是类型*A*的子类型（等价地，类型*A*是类型*B*的超类型），则每个属于类型*B*的值也属于类型*A*，因此对属于类型*A*的值适用的操作和约束也同样适用于属于*B*的值（也会存在适用于属于*B*的值的操作和约束，但是并不适用于只属于*A*的值）。我们区分了单一继承和多重继承（但是只讨论了单一继承），还有标量继承、元组继承和关系继承（但是只讨论了标量继承），同时我们引入了类型层次结构的概念。我们还定义了真子类型和真超类型，直接子类型和直接超类型，以及根类型和叶类型。同时我们描述了不相交假设：类型*T1*与*T2*是不相交的，除非一个是另一个的子类型。做为这个假设的一个推论，我们说每个值只有一个唯一的最确切类型（但是它不必是叶类型）。

接下来，讨论了（包含）多态性和（值的）可置换性的概念，它们都是基本的继承概念的逻辑推论。区分了包含多态（与继承有关）和重载多态（与继承无关）。还展示了包含多态是如何带来代码重用的——借助于动态联编。

然后又开始考虑继承对于赋值操作的影响。基本的想法是不会发生类型转换——值在被赋给一个非确切类型的变量时可以保持其最确切类型——因此一个声明类型为*T*的变量可以有一个最确切类型是*T*的任意子类型的值（同样地，如果定义操作*Op*的结果的声明类型为*T*，则调用*Op*所得到的实际结果的值的最确切类型可以是*T*的任意子类型）。因此我们模型化一个标量变量*V*——或者更一般地，一个任意的标量表达式——成为一个形如*<DT,MST,v>*的有序元组。其中*DT*是声明类型，*MST*是当前最确切类型，*v*是当前值。我们引入了TREAT DOWN（类型下移）操作，使得可以对这样的表达式进行操作，这些表达式在运行时候的最确切类型是其声明类型的真子类型——否则的话，这种操作会在编译的时候报出类型错误（运行时的类型错误也会出现，但是错误只会出在TREAT DOWN操作内）。

接着更细致地探讨了选择子操作。调用类型*T*的一个选择子操作时，有时候会产生一个属于*T*的某个真子类型的结果（至少在我们的模型里是这样，虽然——典型地——在今天的商用产品中并不是这样）：约束特化。然后又比较详细地讨论了THE_伪变量。既然它们只是一些缩写，则约束特化和约束概括在对THE_伪变量进行赋值时都会出现。

然后开始讨论子类型和超类型对等值比较以及一些关系操作（连接、并、交、差）的影响。我们也引入了一些类型检测操作：IS_*T*、IS_MS_*T*，等等。其后考虑了关于只读操作和修改操作、操作版本、操作签名的问题，并指出为一个操作定义不同版本的能力使得对该操作改变语义成为可能（但是在我们的模型中禁止这种改变）。

最后，我们考查了“圆真的是椭圆吗？”这样一个问题。通过考查确立了这样一个立场，即继承只适用于值，而不适用于变量。更准确地说就是，只读操作（只适用于值）可以百分之百地得到继承而毫无问题，但是修改操作（适用于变量）只能有条件地得到继承（我们的模型在这方面与大部分其他的方法不一致。那些方法典型地是要求修改操作可以被无条件地

继承，但是它们接着就会被各种问题所困扰，这些问题自然与所谓“非圆形的圆”等类似的说法有关）。于是据此得出结论——从我们的立场上，约束特化是在逻辑上定义子类型的唯一有效方法。

练习

19.1 给出下列词语的定义：

代码重用	真子类型
哑类型	根类型
约束泛化	动态联编
直接子类型	标量类型
叶类型	签名
非标量类型	约束特化
多态性	可置换性

19.2 解释TREAT DOWN（类型下移）操作。

19.3 区分下列概念：

a. 参数 与 参量
b. 声明类型 与 当前最确切类型
c. 包含多态 与 重载多态
d. 调用签名 与 描述签名 与 版本签名
e. 只读操作 与 修改操作
f. 值 与 变量

19.4 参照图19-1的类型层次结构，有一个属于类型 ELLIPSE的值*e*。*e*的最确切类型是ELLIPSE或CIRCLE，那么*e*的最不确切类型是什么？

19.5 任意给定的类型层次结构包括很多子层次结构，它们仅凭自身也可以被认为是完整的类型层次结构。比如，对于从图19-1中（只）删除了类型PLANE_FIGURE、ELLIPSE和CIRCLE得到的层次结构而言，凭借其自身的特征也可以认为是一个类型层次结构。同样地，（只）删除了类型CIRCLE、SQUARE和RECTANGLE之后所得到的也是一样。但是（只）删除ELLIPSE后得到的层次结构凭借其自身的特征就不能认为是一个类型层次结构（至少没有人可以从图19-1中派生出它来），因为类型CIRCLE“丢失了它的一些继承特征”，而这是它在那个层次结构中本应该有的。那么在图19-1中总共有多少个不同的类型层次结构呢？

19.6 利用本章给出的简要语法，给出类型RECTANGLE和SQUARE的类型定义（为了简单起见，假设所有矩形的中心都在原点，但是并不假设所有的边都是垂直或水平的）。

19.7 根据你对练习19.6的答案，定义一个操作，可以使一个特定的矩形绕它的中心旋转90°，还要给出该操作针对正方形的显式特化。

19.8 下面引用19.6节的一个例子：“关系变量*R*有一个声明类型为ELLIPSE的属性A，我们希望通过查询R得到这样的元组，即A的值实际上是一个圆而且其半径大于2。”下面是在19.6节给出的这个查询的公式：

```
R : IS_CIRCLE ( A ) WHERE THE_R ( A ) > LENGTH ( 2.0 )
```

a. 为什么我们不能简单地在WHERE子句中表达类型检查条件？——比如：

```
R WHERE IS_CIRCLE ( A ) AND THE_R ( A ) > LENGTH ( 2.0 )
```

b. 另一个候选的公式是：

```
R WHERE CASE
           WHEN IS_CIRCLE ( A ) THEN
                THE_R ( TREAT_DOWN_AS_CIRCLE ( A ) )
                                  > LENGTH ( 2.0 )
           WHEN NOT ( IS_CIRCLE ( A ) ) THEN FALSE
      END CASE
```

这个公式会有效吗？如果不会，为什么不会？

19.9 文献[3.3]建议支持这样的关系表达式

```
R TREAT_DOWN_AS_T ( A )
```

这里*R*是一个关系表达式，*A*是该表达式所代表的关系——比如说叫*r*——的一个属性，*T*是一个类型。*A*的声明类型*DT*(*A*)必定是*T*的超类型（这是编译时做的检查）。整个表达式的值被定义为一个关系，而且：

a. 其属性名称与*r*的一样，除了其中属性*A*的声明类型为*T*；

b. 其内容包含与*r*一样的元组，除了元组中属性*A*的值被下移成了*T*。

但是，这个操作还是一个缩写——为什么？请具体说明。

19.10 形如*R*:IS_*T*(*A*)的表达式同样是一个缩写——为什么？请具体说明。

参考文献和简介

19.1 Malcolm Atkinson et al.: “The Object-Oriented Database System Manifesto,” Proc. First International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan (1989). New York, N. Y.: Elsevier Science (1990).

关于一个好的继承模型缺乏一致性的问题（在19.1节提到过），这篇论文的作者这样说：“至少存在四种类型的继承：置换继承、包含继承、约束继承和特定继承……现有系统和原型各自不同程度地提供了这四种类型的继承，而我们并不规定一种特别形式的继承。”

这里还有一些其他的引文表达了同样的观点：

- Cleaveland [19.4]说：“［继承可以］基于［各种］不同的标准，而且并不存在普遍接受的标准定义”——并且给出了8种合理的解释。（Meyer [19.8]给出了12种）。
- Baclawski和Indurkhya [19.2]说：“［一种］编程语言［只是］提供一组［继承］机制。这些机制显然限定了用这种语言可以做什么，以及可以实现什么样的继承视图……它们自身并不能使这样或那样的继承视图生效。类、特化、泛化和继承性只是概念，而且……它们没有一般意义上的客观含义……这［一事实］暗示继承性如何并入一个特定的系统取决于［这个］系统的设计者，这就成了一种要由有效机制来实现的策略选择了。”换句话说，根本就没有模型！

19.2 Kenneth Baclawski and Bipin Indurkhya: Technical Correspondence, *CACM* 37, No. 9 (September 1994).

19.3 Luca Cardelli and Peter Wegner: “On Understanding Types, Data Abstraction, and

Polymorphism," *ACM Comp. Surv*. 17, No. 4 (December 1985).

19.4 J. Craig Cleaveland: *An Introduction to Data Types*. Reading, Mass.: Addison-Wesley (1986).

19.5 C. J. Date: Series of articles on type inheritance on the *DBP&D* website *www.dbpd.com* (first installment February 1999).

对于本章所描述的、并在文献 [3.3]**中得到更规范定义的继承模型的一个扩充教程（带有年代的注解）。**

19.6 C. J. Date and Hugh Darwen: " Toward a Model of Type Inheritance," *CACM* 41, No. 12 (December 1998).

这个简短的说明总结了我们的继承模型的主要特征。

19.7 Nelson Mattos and Linda G. DeMichiel: " Recent Design Trade-Offs in SQL3," *ACM SIGMOD Record 23*, No. 4 (December 1994).

这篇论文给出的理由说明了，为什么 SQL3**的设计者做出不支持类型约束的决定（这是基于早些时候** Zdonik**和** Maier**在文献**[19.11]**提出的观点）。但是我们并不同意这一理由。它的根本问题在于不能正确地区分值和变量（见练习** 19.3**）。**

19.8 Bertrand Meyer: " The Many Faces of Inheritance: A Taxonomy of Taxonomy," *IEEE Computer* 29, No. 5 (May 1996).

19.9 James Rumbaugh: " A Matter of Intent: How to Define Subclasses," *Journal of Object-Oriented Programming* (September 1996).

在19.9**节中提到，我们的观点是约束特化是在逻辑上定义子类型的唯一有效方法。因此，可以注意到这样一件非常有趣的事情，对象世界恰恰是站在完全相反的立场上！—— 或至少是那个世界中的某些人是这样做的。用** Rumbaugh**的话说："** SQUARE**是** RECTANGLE**子类吗？……拉伸矩形的** *x***轴是一件很正常的事情，但是，如果你对一个正方形这样做，则它就再也不是一个正方形了。概念上，这并不是一件坏事。当你拉伸一个正方形的时候，你得到了一个矩形……但是……大多数面向对象的语言并不希望对象改变它们所属的类……所有这些都暗示了（一条）分类系统的设计原则：子类不能通过对父类的约束来定义 "。注意：像在第** 24**章所解释的，对象世界中类这个词所表达的意思经常和我们所说的类型所表达的意思是一样的。**

我们惊奇地发现， Rumbaugh**站在这个立场上显然只是因为面向对象语言 " 不希望对象改变它们所属的类 "。而我们宁愿首先保证模型的正确性，然后再考虑它的实现问题。**

19.10 Andrew Taivalsaari: "On the Notion of Inheritance," *ACM Comp. Surv*. 28, No. 3 (September 1996).

19.11 Stanley B. Zdonik and David Maier: "Fundamentals of Object-Oriented Databases." in reference [24.52].

部分练习答案

19.3 **我们只考虑** f **的情况（值与变量），因为这是一个基本的问题，而在书的其他地方也没有进行明确的讨论（下面的定义取自于文献** [3.3]**）。**

• 一个值是一个 " 单独的常量 "（比如，单独的常量 " 3 "）。一个值在时间和空间上没

有定位。但是，一个值可以通过某种编码方式在内存中进行表示，当然这种编码方式在时间和空间上是有定位的（见下一段）。要注意，由定义可知，一个值是不能被修改的。因为如果可以修改的话，则经过这样的修改以后它就不再是原来那个值了（在一般意义上）。

- 一个变量是一个值的编码的容纳者。一个变量在时间和空间上是有定位的。当然与值不同，变量是可以修改的，即变量的当前值可以被另一个值所取代，而且基本上是与前一个值不同的（当然，修改后的变量还是原来的变量）。

顺便提一句，理解这样一点非常重要，即并不仅仅是类似整数“ 3 ”这样简单的情况才是合法的值。相反，值可以有任意的复杂度。比如，一个值可能是一个数组、一个栈、一个列表、一个关系、一个几何点、一个椭圆、一条X射线、一份资料、一个指纹，等等。当然类似的说明也适用于变量。

19.4 在图19-1中任意类型的任意值的最不确切类型当然是 PLANE_FIGURE。

19.5 22（这个数字包括空层次结构）。

19.6 对所有中心在原点的矩形来说，一个给定的矩形 ABCD可以用任意两个相邻的顶点 —— 比如说A和B —— 来唯一地标识；而一个给定的正方形可以用任意一个顶点 —— 比如说A —— 来唯一地标识。为了把事情讲得更清楚，让 A是在平面的第一象限（$x \geq 0$，$y \geq 0$）的顶点，而 B是在第四象限的顶点（ $x \geq 0$，$y \leq 0$），则我们可以如下定义类型 RECTANGLE和SQUARE：

```
TYPE RECTANGLE POSSREP ( A POINT, B POINT ) ... ;
TYPE SQUARE POSSREP ( A POINT )
     CONSTRAINT ( THE_X ( THE_A ( SQUARE ) ) =
                - THE_Y ( THE_B ( SQUARE ) ) AND
                  THE_Y ( THE_A ( SQUARE ) ) =
                  THE_X ( THE_B ( SQUARE ) ));
```

19.7 下面定义的操作是特定的修改操作。作为一个辅助练习，定义与这些操作类似的只读操作。

```
OPERATOR ROTATE ( R RECTANGLE ) UPDATES ( R )
    VERSION ROTATE_RECTANGLE ;
    BEGIN ;
         VAR P POINT ;
         VAR Q POINT ;
         P := THE_A ( R ) ;
         Q := THE_B ( R ) ;
         THE_X ( THE_A ( R ) ) := - THE_Y ( Q ) ;
         THE_Y ( THE_A ( R ) ) :=   THE_X ( Q ) ;
         THE_X ( THE_B ( R ) ) :=   THE_Y ( P ) ;
         THE_Y ( THE_B ( R ) ) := - THE_X ( P ) ;
         RETURN ;
    END ;
END OPERATOR ;
OPERATOR ROTATE ( S SQUARE ) UPDATES ( S )
    VERSION ROTATE_SQUARE ;
    RETURN ;
END OPERATOR ;
```

19.8

a. 这个表达式在编译时会出现类型错误，因为 THE_R需要一个类型为CIRCLE的参数，而A的声明类型是ELLIPSE不是CIRCLE（当然，如果在编译时没有做类型检查，那么一旦在运行的时候，我们遇到一个 A的值是椭圆而不是圆的元组，就会出现运行时的类型错误）。

b. 这个表达式是有效的，但是它会产生一个属性与 R一样的关系，却不是一个属性 A的声明类型是CIRCLE而不是ELLIPSE的关系。

19.9 这个表达式是如下形式表达式的缩写

```
( ( EXTEND R ) ADD ( TREAT_DOWN_AS_( A ) ) AS A' )
                        { ALL BUT A } ) RENAME A' AS A
```

（A'是任意一个名字，只要它与处理关系 R得到的结果关系的任何一个属性名都不一样。）

19.10 这个表达式是如下形式表达式的缩写

```
( R WHERE IS_( A ) ) TREAT_DOWN_AS_( A )
```

而且后一个表达式本身也是一个更长的表达式的缩写，即我们在练习 19.9的答案中所看到的表达式。