计算机科学1991№.3

计算反射和面向对象程序设计(I)*)

奚建清　胡守仁　　（国防科技大学计算机系）

摘　要

Computational reflection is a kind of programming technique with increasing importance and understanding. This paper discusses its general idea and implementation technique. First the human's counterpart—reflection—is briefly discussed. Then a general model of computational reflection is given with the discussion of some problems and their possible solution. At last more examples show the usage of computational reflection in software systems.

一、引　言

计算反射（computational reflection）是80年代初发展起来的程序设计概念和技术，早在这个概念提出以前，人们在编写程序如compiler、debugger等时就已隐含地使用了这种技术，其后计算反射的思想越来越受到人工智能、知识工程等领域的重视[2,3,4]，面向对象程序设计的兴起和发展为计算反射的概念和技术的进一步成熟和发展开辟了更好的前景，许多文章都论述了在不同计算模型中的反射行为，但未给出较严格的定义，本文第一部分首先简单地讨论了人的反思行为，然后在[1]的基础上建立了计算反射的一般模型，它具有较好的普遍性，据此分析了反射的能力。结合这个模型讨论了在具体计算系统中的应用例子，第二部分（连载）介绍在面向对象程序语言和知识库模型中计算反射思想的体现和实现技术。

二、计算反射的一般思想

Reflection本来用于人的反思现象，故这一节先简单地讨论人的反射思维，然后讨论一般的计算模型。

1. 人的反射思维

Reflection一词也可以译为反思或反省。其意义：一是对自己过去思想（包括自己的思维或行为过程、所认识到的环境等）的考虑，二是与自己商讨。这两种意义有共性，即有下面几方面的特点：

1）反射是一种思维过程。

2）自身的思维过程（包括记忆状态）是反射思维过程中的被思维对象。

人能认识、回忆自己的思维过程或行为，否则就不能实现反思。当然实际上任一时刻的思维不会把这个时刻的思维作为思维对象，因为这会导致思维的无限循环。下面是几点关于人的反思思维的认识，对理解计算反射是有帮助的：

*1　思维过程是多种多样的，如推理、回忆、类比和灵感等，但这些思维活动是人脑中神经网络的活动的宏观效应，反射思维的对象是这些宏观的活动，而不是细微的神经元的活动，因为人不能认识到他（她）思维时的具体神经元的活动。

*2　人经常有意或无意地利用反射思维来反省自己的行为和思维方法，这样才能不断地改正、完善自己，积累经验，从而对自

*）本研究项目受到国家自然科学基金的支持

己的反射思维的前提是能认识到自己的思维行为、习惯和自己所处的环境，即有自我的意识，对自己认识得越多，则越有条件对自己的思维进行反思，但人对自己的认识还是有局限的。

或许是受到了人的反射思维的启发，才为计算系统的类似行为定义了类似的术语。

2. 计算系统的反射行为

这里用[2，3，4]中给出的关于计算反射的描述：计算反射是反射系统呈现的一种行为，这个系统是一个计算系统，它能因果关联地对自身进行处理。Maes接着对上述定义中的关键词作了说明，'计算系统'是指基于计算机的对一问题域进行处理(询问及操作)的系统，'因果关联地'是指计算系统中一个问题域对应的表示结构和这个问题域的联系是直接的，亦即表示结构的改变会直接导致问题域的相应改变，反之亦然，从而计算反射系统对自身的处理将直接导致整个计算系统的改变。

可是Maes及其它人并没有对计算系统的'自身'的概念作详细的说明，而理解它们对于理解反射是很重要的。我们说一个计算系统包括了计算机硬件，程序和数据。例如一旦给定了用户程序，则计算系统的行为就由计算机硬件、操作系统、编译程序或解释程序，以及用户程序和输入数据决定。输入数据之所以也决定系统的行为，是因为它不仅影响了本次计算的行为，而且输入数据的历史也影响了具有演化功能的系统的行为。从而我们可以定义计算系统是由硬件、程序和数据组成的（数据作为系统的一部分也说明了自身中也包括了周围世界的一部分，而不是封闭的系统）。

即使定义了计算系统自身的概念，但还不能清楚地定义反射的概念，因为反射是一种行为，而计算系统的所有行为并不完全都是反射行为，故有必要对计算系统的行为作一番分析。

为了分析这种行为，我们采用[1]中的过程反射系统的定义。一个过程反射程序设计语言是一个特殊的反射结构，其中所有程序的执行并不是由一个原始的且不可访问的解释器的执行机构来完成，而是通过一个表示这个解释器的程序的明显的运行来执行的，这个程序叫做反射处理器程序(RPP)。如果假设不同的语言的名空间是不相交的，则要求RPP的语言和用户语言是一样的，从而存在一个RPP的拷贝的无穷序列，每个RPP均在执行下一层的RPP或用户程序（图1）。

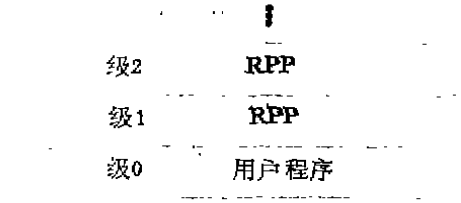

图1 无限的程序级组成了计算系统

这个结构与元循环（meta-circular）解释器是很类似的。本质区别是低级程序可以有程序段运行在上一层，即这个程序的解释器一级甚至更高级上，从而可以明显访问以前对这个程序是透明的那些计算信息。对这些信息的访问构成了过程计算模型中反射的基础。

这个模型使我们能较为方便地认识和讨论计算反射，可是应注意到，正如Maes对计算反射的描述以及我们对人的反思行为的观察那样，计算反射的概念并不是以计算系统的分级概念（例如分成目标级或用户级以及元级或解释器级(为前提的，这种与元级概念的混淆易使人误认为计算反射仅仅是元级概念的一种扩充或发展。由于目前的专家系统、知识库等程序系统一般是分级的，从而这些系统中的反射模型的设计也是以分级为基础的。诚如人虽然能抽象关于自然界的知识和关于自身的知识，但不管是获取过程或是在大脑的记忆状态，还是在使用过程中，我们并不认为知识中存在着级别，而是存在着联系，它们是纵横交错在一起的。可是分级的模型不仅符合当前一般计算系统的实际情

况，而且能大大简化讨论，故我们还是遵循着图1中的模型来展开讨论。

3. 计算反射的一般模型

我们从图1中的结构出发进一步分析程序的性质。一个程序的静态状态是其代码体（包括定义环境），代码体由若干函数组成。动态状态是其静态状态加上变量空间（输入数据可以看成是变量空间的一部分）以及当前执行点。定义变量元组Sp=〈BODY VA VD F CONT〉为程序（段）P的动态局部状态，假设这些状态变量由程序P的解释器在适当时刻定义，其中变量BODY是P的代码体，在程序开始被执行时定义；变量VA是P中所有当前已访问到的变量名字的集合，在P的运行中当访问第一个变量时被定义；变量VD是P中到执行点为止已定义的且没有被删除的变量名字的集合，在P的运行中当定义第一个变量时被定义；变量F是P中到执行点为止已定义的且没有被删除的函数（子程序）名的集合，在P运行中当定义第一个函数时被定义；变量CONT是P执行时的运行位置指示，在程序开始被执行时定义。当然这些局部变量也可以在其它适当的时刻定义。这些变量存在的必要条件是这个程序的存在。于是图1中每个级别的程序均对应有一个动态局部状态来表示它们，它们均由上一级程序定义，这些状态组成了一个状态序列S0,S1,…。其中S0是用户程序的局部状态，Si是第i级Pi(RPP)的局部状态(i>0)，要注意到变量VD和VA的值是动态的，即和对应程序的运行直接相关。这个状态系列刻画了对应计算系统的任一时刻的'自身'。

当然我们还可以定义其它局部变量，如关于程序的输入参数和输出参数的类型，程序的产生者，版本等变量。为简单起见，我们仅讨论上述反映程序主要状态和结构的几个变量。称这个计算模型为一般模型。

在这样的系统内，一旦某一变量被定义或被引用，则应将它的名字放入相应的VD或VA中。一变量访问必须后于它的定义或说明。程序中对一个变量的访问是通过一个从变量名字到变量本身的映射函数（如Hash查找函数）来实现的，如果变量名字即为变量，则这个函数就是恒等函数。

对像图1中这样的计算系统，假设不同级上的名空间是不相交的，且需访问某一级定义的状态变量的程序（段）均应运行在那一级上。我们把图1中在i级上的程序记为Pi，Pi的状态变量（由Pi-1定义）记为〈BODYi VAi VDi Fi CONTi〉，(i≥0)。如果Pi对应的程序被提级而运行在更高级上，则这个程序的标号仍为Pi，至此我们可定义

Pi是反射的，如果存在Pj (j>i)，使得VDj∩VAi=φ；一个计算系统是反射的，仅当其中某一级的程序是反射的。

一个一般模型如果是反射的，则称为一般反射模型，亦简称为一般模型。

注意这个定义要求一般模型中的计算反射符合两个条件：一是低级的程序运行在高级上，二是低级的程序要访问高级别的程序定义的状态变量。在上述模型中我们更感兴趣的是用户程序是反射的那些计算系统，而且不失一般性，本文主要考虑能访问上一级程序定义的状态变量的反射行为。

我们规定如果对VA的操作引入了新的变量名（从而引入了新的变量），则这些变量名属于和这个VA同级的VD，即把它们看成是由下一级程序定义的变量。

由于在过程性语言中赋值操作实现了变量值的修改，用户程序可以用赋值操作来访问和修改高级程序定义的状态变量，从而一个语言的变量实现和赋值操作的语义对于反射能力是至关重要的（我们不考虑通过函数调用的虚实参数结合实现的访问）。下面分析各种变量实现与赋值操作的语义和对应的反射能力或限制。

4. 赋值操作与反射能力

所谓反射能力是指某一级程序可以操作上级程序定义的变量（主要指局部状态变量）的能力，对上级变量的操作可能会导致错

误。一是可能使程序无法执行，例如一旦删除了上级的BODY变量或对其值作了修改，则本级的程序（可能）不能正常地由上面级别的程序解释运行，因为可能发生语法或语义的错误（详见下面）。这种错误可以通过语法或语义检查来识别。第二种错误是指当修改上级的VD或VA变量时，可能使得一些变量丢失即删除（相当于收回其地址），如果系统采用拷贝方式赋值，并且变量和变量名是不区分的，则被删除的变量又可能出现在系统的其它地方，这种错误是系统语法和语义检查不能识别出来的，称这种错误是变量悖论。下面主要讨论一般反射模型中可能避免的变量悖论等错误，限于篇幅我们不考虑对BODY、CONT等变量的修改。

情形1 就像Common Lisp，Scheme等语言一样，我们定义一个变量是一个名字和常量的联结，从而一个变量名字和变量本身是不一样的。变量名仅仅是变量本身的一个标志符，因此变量名的存在并不意味着变量本身的存在。

又规定只能让变量名出现在程序中，亦即变量名而不是变量本身出现在表达式或语句中。在定义或说明一个变量时，除非指定了初值，否则不认为新的变量被引入了系统，而仅认为一个变量名被引入了系统。对变量的访问是通过变量名来实现的。当VAi的值被改变时，若某个（些）来原来VAi中出现的元素在新的VAi中不出现，并且这个（些）元素属于VDi，则据上面的假设，这个（些）元素被删除，原来VAi中的其它元素不删除仍能在Pi中被访问。在这样的系统内，我们可证明对计算系统中的所有变量的操作均不会出现变量悖论错误。

因为如果Pi删除了它定义的一般变量a，则从VDi和VAi中因果相关地将a的名字删除。如果其名字出现在系统的其它地方，则由于对这个变量的访问要经过一映射查找函数，故系统能检查出这个变量不存在了。若这个变量是上级定义的VD或VA变量，则据上面的假定，本级程序不能删除这种变量。

情形2 假设变量和其名字是不区分的（但变量必须有值）。于是在这种系统内，变量可以作为值或集合值的一个元素出现在表达式或语句内，例如VA或VD的值中的元素可以是变量。如果系统采用拷贝方式赋值，则A:=VD或A:=VA的操作使得VD或VA中的同一变量重复出现在系统的多个地方，从而对VD或VA的修改可能导致变量悖论错误，所以在这种系统内，如果某个VD或VA变量的（部分）值被其它变量所拷贝共享，则这个变量不能被删除、或修改值。

情形3 现在考虑可以让变量存在但处于无定义状态的系统。对任一变量x，x:=⊥表示x为无定义。对于VAi:=⊥的操作，亦即让VAi处于无定义状态（注意其值不是为空集），则Pi访问一个新的变量或删除一个变量的操作均维持VAi的无定义状态，（因为这两个操作均对VAi的值进行修改），从而导致系统的不一致性。对于VDi变量也有类似的结论。所以在这样的系统内应禁止任意的VA和VD变量的值为无定义，但加适当的控制，例如在过程语义上将VD或VA的无定义状态看成是值为空集就可解决这个问题。

情形4 分别在上述情形1、情形2和情形3中，如果在对VAi或VDi的值的修改时下级程序可以删除上级程序定义的变量（如VAi，BODYi），则这种删除可能导致整个（子）系统的破坏（详见下面对变量BODY和F的讨论）。

5. 一个实际的计算反射模型

图1中描述的情形是一种假想的计算模型，实际中一个有限程序的反射深度（即它需要运行的最高级别，在[1]中称为反省度）是有限的，大于这个深度的RPP仅在执行RPP本身，而不是用户程序。所以只需要有穷的解释器级别来执行用户程序，且而为了在计算机硬件上实现这个计算模型，必定有一个解释器是用其它语言（如机器语言）写的，从而这个级别定义的状态变量一般不能

由下级的程序直接访问。可是由于用户程序的反射深度在实际执行这个程序前是不能预先知道的，故不能预先由用户程序的反射深度来确定计算系统的模型，〔1〕中反射语言3-Lisp的设计中引入了级别升降(Level-shifting)的重要思想来解决这个问题。其主要思想是：

如果某一级运行的RPPi能发现当前它所运行的程序Pi-1是反射的，即需要由上一级甚至更上级的RPPj来处理（i<j），那么如果由RPPi定义的状态变量值能推出RPPj定义的状态变量的值，则可由RPPj从计算出来的状态出发接着来运行Pi-1，并且相当于RPPj是从头开始一直运行的。

这样只要有一个实际实现的解释器G′，它可以是由机器语言实现、或是由CPU实现，也可以由其它语言实现。用户程序的执行分成三种情形，一是如果用户程序是直接实现的（即解释器G′中有对应的程序段），则由G′直接执行，例如Lisp中的函数调用(+12)、Prolog中的询问Less-than(2，3)（如果算术已经在Prolog解释器中实现）均可直接计算。第二种情形是如果用户程序是由其它已直接实现的子程序（函数）复合而成，则可以先分解后执行各个可执行的子程序。例如Lisp中的(second 2 a-list)的计算、Prolog中在目标级的搜索。第三种情形是用户程序必须提高一级或数级执行，则由G′产生相应的RPP的状态并将用户程序交给这个RPP执行，而G′执行这个RPP。

从而G′必须既能执行用户程序，也能执行RPP程序。可是为了提高效率，没有必要由RPP运行的程序应尽量由G′来直接执行。所以一旦发现用户程序不是反射的，即（经分解后）可由G′直接执行，则应该降级到用户级由G′来直接解释执行。

用级别升降的思想来实现反射是分级计算模型的经典作法。下一节中的例子都采用这种方法。

三、计算反射在实际系统的例子

这一节主要介绍把计算反射推广到基于规则的计算系统中，用例子说明计算反射在这些系统中的应用，结合这些例子简单地分析一下相应系统中计算反射的问题。

例1 一个基于规则的计算系统例子

```
WORKING-MEMORY ((cost(a)=5 True)(haz(a)<0.3 True) (cost(b)≤2 True)(haz(b)>0.5 True)(01 True)(p1,)(p3,)(p2,)…)
GOAL, ((p1, True)(p3, True))
RULE MEMORY : (1)IF cost(a)<6 and haz(a)<0.35 and 01
    THEN using(a)and set (p1,True) and set(p2, False)
    (2) IF cost(b)<3
    THEN using(b)and set (p2, True) and set (p1, False)
    ...
META-RULE MEMORY, (3)IF error-flag-1 and haz(x)<0.4 and cost(x)<6
    THEN set(rule-to-be-fired (rule-involving (x) ), True ) and set ( data-elm (o1) false)
    (4) IF satisfied(1) and satisfied(2)
    THEN set (error-flag-1, True)
```

这个例子中，目标级中同时满足条件而可以点火的规则结果之间是矛盾的，为了解决这个问题，系统将进入元级运行META-RULE。元级规则将通过分析目标级的规则和数据的状态并据优先使用安全的材料的原则来确定选择那一个规则点火。本例中元级选择使用材料b的目标级规则。

和过程模型相比，用户程序在这里相当于目标级和元级规则。在上述META-RULE中，函数data-elm相当于VD（或VA）变量，rule-to-be-fired类似于变量CONT，这些变量都是解释器一级定义的变量，解释器控制了由目标级到元级的转换以及元规则结果对目标级的影响。

在上述产生式系统内，如果数据元素的值可以为WM或其一部分，则对这种元素的修改可能导致变量悖论。（转第43页）

基于知识的决策支持系统KB-DSS的原理及实现方法

刘鲜京　　（山东省计算中心）

摘　要

本文简要介绍利用专家系统的概念、实现方法与决策支持系统的开发技术相结合的一种基于知识决策支持系统KB-DSS的结构设计及其特点，着重阐述了KB-DSS的核心部份——问题处理子系统PPS的设计思想。

决策支持系统（DSS）的概念自1971年提出后，现已有了很大的发展。目前DSS大致分为以下两大类：

1.基于算法模型的结构。它是一种基于常规数据处理系统的DSS，此类系统中设有一个（或多个）模型库，当用户进行决策操作时，可根据对问题的了解，选择合适的算法模型；或通过会话系统（LMS）和问题处理系统（PPS）对用户的问题进行判断，自动选择合适的模型对问题进行处理。该结构常用的算法模型有：优化过程、数字数据、目的-手段的灵敏度分析、统计分析、风险

（接第22页）

例2 两个反射的产生式规则

1. IF it is not known whether a data-element to be true, THEN it is definite(1.0)that the data-element is false.
2. If currently seeking rules about data-element a, look first at rules that mensiona somewhere inside them; look next at rules that mension some generalization of x.

第一个规则使用封闭世界假设（CWA）来处理不完全的（关于某个数据元素的）知识，但规则使用CWA的前提是对其所在库中知识的了解亦即需要访问库的状态。上述规则涉及的状态是由working memory（WM，相当于VD）构成。

第二个规则涉及到对当前目标和规则库的访问，并指导目标级的规则的选择。

四、讨　论

从上面对反射的定义和例子看，计算反射提供了无反射能力的系统无法提供的功能，即程序更具灵活性和适应性。一旦"自身"的许多信息能够被因果关联地表达和计算，则程序对自身的访问和修改的能力使程序的应用面更广。另外，利用修改自身能产生出具有一定程度的自适应的程序，从而这种程序即可以有学习的能力，能高效地处理新的问题域。这种能力在支持动态修改的面向对象程序设计中的优点更为明显，我们将在第二部分讨论这些。

类似于人的反思能力是有限的，计算反射的能力并不是无限的。正如在第二部分中看到的那样，低级程序只能访问到上级程序定义的（部分）状态变量，而且一般模型中的这些状态变量（如VA）并不都在实际系统中定义。Maes把这种现象称为反射的理论相关性，即能反射的是系统明显给出的关于自身的描述，这些知识是有限的，而隐含的以及未涉及到的有关知识可能是无穷的。关于计算系统的明显知识越多，可开发的反射力就越强。