一个基于广义函数机制的反射结构*

赵银亮

(西安交通大学计算机系,西安 710049)

摘要 本文讨论了一个以扩充Lisp方式实现的CLOS系统中的反射结构及其实现技术,该反射结构基于统一的对象表示,支持广义化对象模型,与宿主语言的反射有一致性.其中的反射计算采用了CLOS的元对象协议的思想.

关键词 计算反射*,面向对象的程序设计,解释程序.

计算反射是指一个计算系统靠自身的计算来修改自身同时又作用到该计算中的能力.在一个具有反射结构的OOP语言中,语言的解释器本身的状态和功能随着解释执行该语言程序发生变化,并同时又作用到对该程序的解释中.从使用角度而言,按照语言提供的OOP范例修改语言的解释器,从而构成新的OOP范例.这种解释器需引入内部数据表示对象、引入程序表示关于这些对象的处理(数据+程序=解释器的自表示),自表示与语言的OOP范例之间有因果关联(任一个发生改变即导致另一个变化),解释器提供按语言的OOP范例修改其自表示的功能.

典型的计算反射的研究见文献[1,2]等,其反射结构的共同点是引入元对象来共同表示解释器和用户程序,其中用元对象上的方法来实现与其对应的对象,这些方法是可访问的和可修改的.另外,元对象的内部结构也是可访问的和可修改的.分别归纳为:(1)以类刻划对象结构,以元对象刻划对象行为的模型[1];(2)类兼作元对象的模型[2].这些模型的局限性是基于单一的消息发送机制,不支持方法组合和多重方法.

CLOS系统是一个具有多重方法和多重继承,以广义函数机制代替消息发送机制的OOP系统,其中用户接口部分已成为Lisp方面的一个标准OOP语言[3],而CLOS的元对象协议(MOP)提出了用广义函数实现CLOS系统以支持反射计算的思想,目前这方面的工作仍在研究之中.我们在Lisp中扩充实现了一个CLOS系统[4],采用MOP的思想建立反射结构,从对语言特征具体化入手研究了MOP设计及在反射结构中的作用,本文着重讨论了这种反射结构的实现方法,并进行了分析.

1 反射结构

就具有反射结构的语言而言,其反射功能从语言的各个侧面反映出来,OOP语言的反

* 本文1992-07-03收到,1993-02-18定稿

作者赵银亮,1960年生,讲师,主要研究领域为人工智能,程序设计.

本文通讯联系人:赵银亮,西安 710049,西安交通大学计算机系

射功能从对象级表现出来,混合型 OOP 语言的反射功能也保留了宿主语言级的反射功能.理论上总可以把这种语言的解释器的功能划分成与反射有关的部分(反射部分)和无关部分(不变部分),前者总是解释器的部分功能,而后者与反射计算透明.反射部分决定语言的反射程度,反射程度大小决定反射结构的好坏.

解释器的反射部分划分成由一组反射模块构成,这些模块同时须具备可扩充性,它们是对语言提供给用户的 OOP 范例的各个侧面的内部表示,在反射计算中,随着解释器的上移(见下面解释),有关的反射模块最终都要退化成非反射代码(属于解释器的不变部分),例如在 Maes 的反射结构[1]中把继承处理,实例生成,对象打印等作为反射模块,并表示成元对象上的方法,在消息发送意义下通过遮挡这些方法获得反射计算.

广义函数机制[3]用于实现多重方法的多重继承和方法组合,比消息发送机制更具有一般性.由于多重方法与多个类(称为类列)有关,不能被封装在单个类中,所以,方法总是隶属于广义函数,方法的继承、调用等都由广义函数来完成.广义函数机制的可扩充性表现在:对类列 T,(1)给 T 定义基本方法遮挡掉原有的或继承下来的方法从而改变 T 的行为;(2)给 T 定义基本方法,其中利用 call—next—method 调用原有的方法,从而扩充 T 的行为;(3)给 T 定义辅助方法访问参数信息或扩充 T 的行为等.基于此特点,反射模块用一组广义函数构成,反射计算都基于这组反射模块进行.而反射程度最终取决于反射模块的表示形式、界面、与其它模块的关系等.

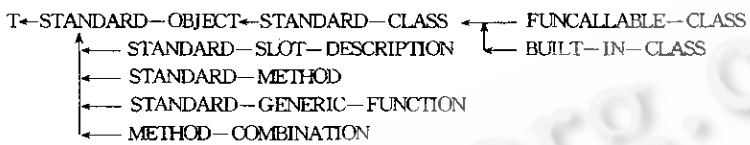
借助于广义函数机制,一个反射模块可以实现成一个广义函数,或者由多个广义函数有机地结合在一起.因此,解释器的反射部分最终表示成一个广义函数的集合(这种广义函数称为反射函数).由于广义函数与类、类的层次结构、以及相关的方法有关,所以,构成反射模块的广义函数要按类、元类组织到一起,这样才能利用语言中提供的 OOP 范例通过 OOP 完成反射计算.与反射模块有关的类作为核心类.

从对象级来看解释器的运行过程,体现了一种无穷反射塔.解释器遇到反射函数时从基级运算转变为元级运算,当在元级运算中又遇到反射函数时又进入元级运算(前者相对于后者为基级),依次类推.上述每一级运算都是由一个解释器完成的.基级运算到元级运算的转变(上移)由实例化关系 I 来控制,类似于 Cointe 的反射结构[2],一般地有:I(实例,类),I(类,元类),I(元类,元类).其中元类中有一个称为不动点的元类,它是任何元类的类.类是实例的模板,又决定实例的行为,与实例有关的基级运算所对应的元级运算是指调用这些实例的类上的方法或访问其中的信息.

按照以上的设计思想,我们首先对解释器的反射部分进行了设计.反射部分的功能尽量包括与 CLOS 的 OOP 范例有关的内容,并按广义函数和核心类方式组织,其语义为由解释器提供的 OOP 范例.

具体而言,把解释器的反射模块表示成广义函数,并把它们及与其相关的类、方法、方法组合对象称为元对象(注,此概念是从语言整体角度定义的,不同于一般的元对象概念),这些元对象及调用界面说明构成一个元对象协议(MOP)并提供给用户,该 MOP 作为对系统自身结构和行为的表示,同时也将是构成用户程序的一部分,与系统的行为(OOP 范例)有因果相关性.用户依据广义函数机制按 CLOS 的 OOP 范例扩充或改变 MOP 实现反射计算.

核心类既是解释器赖以运行的基础又是用户程序借以建立自己的类的层次结构的一个初始情形，事实上，MOP 中的方法都表示成核心类上的方法，而用户定义的类都是对核心类的扩充，用户通过改变这个核心类上的方法或者扩充核心类，有选择地继承、改变核心类上的方法，实现反射计算. 扩充的基础及范围取决于 MOP . 核心类设置如下，其中箭头表示实例化关系.



核心类中元类有 3 个：FUNCALLABLE—CLASS 用于实现可当作函数调用的对象，如广义函数和方法组合等，BUILT—IN—CLASS 用于实现 Lisp 的数据类型到类的转变，STANDARD—CLASS 为不动点元类，包括处理用户定义的对象. 其它核心类用于刻划(实现)槽、方法、广义函数、方法组合对象，例如打印、跟踪、检查、生成实例等.

基于上述核心类我们把解释器反射模块划分成：继承处理、槽存取、实例生成、实例初始化、对象管理、广义函数调用、方法组合、类重定义等，并且每个模块细分为多个子模块，分别对应于一个广义函数.

2 实 现

在具有反射机制的 OOP 语言中，语言的解释器一般设置成一个元循环解释器，如 Lisp 的求值器. 对在 Lisp 中实现的 CLOS 而言，所建立的对象级的反射机制要与 Lisp 的反射机制保持一致，我们扩充 Lisp 的求值器使其成为支持对象级反射机制的基础. 首先，在数据结构上，把 Lisp 的数据类型转变为类，而对象均用 Lisp 的数据对象来实现，这为解释器在目标级上工作准备了前提. 其次，对反射函数和非反射函数在求值器级不加区分，也即求值器本身不对反射函数进行退化处理，由反射函数自己去完成. 这样，不需改变求值器，只要统一对象表示形式，统一广义函数自身处理，即可构造出 CLOS 的解释器.

对象表示为统一的形式，表现在结构和处理两方面，对象的结构由其元类控制，对其如何处理由其类控制. 在结构上不管实例的元类为何其数据结构中都要包括它自己的局部槽块和指向它的类的控制块的指针，控制块是指把类中描述其实例的结构和对实例的访问控制与该类的其它内容独立出来而形成的数据结构. 具体包括访问模式、局部槽名表、共享槽信息和指向类的指针. 以类为例，它的局部槽块中有表示继承的信息、层次关系的信息、类的优先次序表、槽对象、初始化参数、控制块以及相关的方法等，在所指的元类的控制块中描述了上述局部槽块的结构. 元类为 STANDARD—CLASS 的对象表示为一个结构，元类为 FUNCALLABLE—CLASS 的对象表示成一个函数，这两种表示既用于对象级运算又可由非反射函数当作结构和过程加以处理. 按照这种表示，Lisp 的求值器中对常量和变量的处理以及函数返回值控制不必区分是按对象来处理还是按 Lisp 的数据结构来处理，事实上，对象都是按照 Lisp 的数据结构来处理的. 另一个与求值器有关的内容是读入、打印和参数控制，由于 CLOS 中没有规定对象的可读入的外部形式，因此没有对象的直接输入方式，只

可能按内在数据结构输入，这时输入部分不做改变. 对于输出而言，我们给对象规定了一个外部形式，即元类为除 BUILT－IN－CLASS 以外的对象的打印形式都为 #〈类名 实例名 地址〉，为了实现这一点我们把打印模块修改按 print－object 的功能进行工作. 对于函数参数控制按内在数据结构进行，求值器中不做改变.

在实现反射函数时，我们把它构造成一个一般函数，在这个函数中对其实参按照广义函数机制选取可用方法，进行方法组合，构成有效方法并调用之，返回结果. 可以看出，从求值器的角度来看，广义函数与一般函数完全一样. 构造上述要求的广义函数要求宿主语言具备生成函数对象，并且既能当作数据对象又能当作程序对象处理. 另外，考虑到实现效率，我们对广义函数做了静态优化[4]，按惰性修改方式建立广义函数代码，当第一次调用时执行惰性码，该惰性码求实参情形[4]及与其对应的有效方法，构造运行码并加装到广义函数中，然后调用之. 此后再调用这个广义函数时只执行运行码，除非与该广义函数有关的内容产生了修改(这些修改将驱动有关的广义函数重置惰性码). 静态化程度与反射程度成反比，因为反射程度大表示上述驱动因素增加，从而增加运行惰性码的频度，降低效率，而提高静态化程度是为了提高效率.

考虑到解释器用广义函数实现，解释器的生成过程要经过两个阶段完成，前一阶段主要用来构造原始广义函数机制，即不包括辅助方法处理和非标准的方法组合等，使得广义函数的定义、调用能够进行. 后一阶段利用原始广义函数机制逐步建立解释器的反射部分，然后，把原始广义函数机制转变成正常的广义函数机制. 具体过程如下：(1)采用特殊的代码生成核心类，这是实现反射模块的前提；(2)在原始广义函数机制中 defmethod(用于广义函数和方法的分散定义，也用于定义解释器的反射部分)，方法管理函数 add－method、remove－method 等，有效方法查找(method－dispatch)等均利用非反射代码实现，而在正常的广义函数机制中，要换成反射代码，即原始广义函数是把正常广义函数中调用反射函数的地方用非反射代码实现.

3　讨论及结论

本文研究了支持广义化对象模型的反射结构及其实现方法，将语言的解释器划分成反射部分和不变部分，构成反射部分的反射模块在反射计算过程中最终退化成非反射代码运行，反射部分基于核心类建立 OOP 语义，并具体化为核心类上的方法，以 MOP 的形式提供给用户作为进行反射计算的基础.

本文讨论的反射结构中主要存在的问题是效率问题，由于反射程度和静态化程度成反比，而静态化程度是为了提高运行效率，特别是广义函数调用的效率，所以为了保证系统运行效率，要丧失一定的反射程度. 另外，对象级上的反射受到宿主语言级反射的支持，语言的统一性是保证这两种反射一致性的前提.

参考文献

1　Maes P. Concepts and experiments in computational reflection. OOPLSA'87 Proceedings, 1987: 147－155.

2　Cointe P. Metaclasses are first class objects: the OBJVLISP model. OOPLSA'87 Proceedings, 1987.

3　Bobrow D, DeMichiel L, Gabriel R *et al*. Common lisp object system specification X3J13 document88－002R.

SIGPLAN Notices, 1988.

4 赵银亮,郑守淇,常兵等. XJD—CLOS系统实现研究. 第二届中国人工智能联合学术会议论文集,杭州:浙江大学出版社,1992. 44—48.

IMPLEMENTATION OF A GENERIC FUNCTION—BASED REFLECTIVE ARCHITECTURE

Zhao Yinliang

(*Department of Computer Science, Xi'an Jiaotong University, Xi'an* 710049)

Abstract In this paper, a reflective architecture of the CLOS system which is implemented by extending Lisp and its implementation is discussed. The architecture is based on unified object representation, supports multi—methods and method combination, and consists with the host language.

Key words Computational reflection*, object—oriented programming, interpreter.