逻辑程序设计语言 Gödel与 Prolog的比较

昌 杰,赵致琢,李慧琪,高 伟
(厦门大学 计算机科学系,福建 厦门 361005)

摘 要:在多态多类的一阶逻辑基础之上,围绕类型系统、模块系统、控制机制、元程序设计和输入/输出部分对 Gödel语言功能进行分析.重点比较了逻辑程序设计语言 Gödel与 Prolog的联系与区别,通过比较分析,表明由于摒弃了 Prolog语言中的非逻辑成分,引入了多种新的语言成分, Gödel语言具有更好的说明性语义和执行效率.

关键词: Gödel语言; Prolog语言;模块系统;类型系统;控制机制;元程序设计
中图分类号: TP311 **文献标识码**: A **文章编号**: 1673-4785(2009)02-0163-06

A comparison between the logic programming languages Gödel and Prolog

CHANG Jie, ZHAO Zhi-zhuo, LI Hui-qi, GAO Wei
(Department of Computer Science, Xiamen University, Xiamen 361005, China)

Abstract: We analyzed the functions of the Gödel language on the bassis of the first-order logic with polymorphism and multi types. The focus was on its types, module system, control mechanisms, meta-programming, and input/output. Emphasis was on comparisons of relationships and disparities in Prolog and Gödel. The comparisons showed that the Gödel language is more declarative and efficient because it eliminates the non-logical parts of Prolog and introduces many kinds of new language elements.

Keywords: Gödel; Prolog; modular system; type system; control mechanism; meta-programming

Prolog是当代最有影响的人工智能语言之一,由于该语言很适合表达人的思维和推理过程,分别在机器定理证明、自动推理、专家系统、智能规划等许多方面得到了广泛的应用,成为人工智能应用领域的开发工具.但是, Prolog在使用中也暴露了该语言存在的不足:缺乏类型系统、基于 Horn子句的语句形式限制了语言的可表达性能力、控制机制中的非逻辑成份易于导致语义问题等.于是,发展新型逻辑程序设计语言成为一种选择[1].

Gödel语言是继 Prolog之后出现的一个新型说明性通用逻辑程序设计语言,由以英国 Bristol大学的 Lloyd和 Leeds大学的 Hill为代表的研究小组在 20世纪 90年代中期设计开发.由于 Gödel语言是从 Prolog发展而来,两者之间不可避免地存在一些联系和相似之处.比如,在语法结构上使用了项、原子和子句,其中,无条件子句、条件子句和目标子句构成了它们的程序主体.另外,在执行机理上,合一算法和 SLD反驳—消解法为结果的正确性提供了有效的保证. Gödel同时也借鉴了 Prolog和程序设计领域的最新成果,引入类型系统、延迟计算和剪枝操作等诸多新的语言成分,将语言的理论基础扩展到多态多类的一阶逻辑,试图解决 Prolog中程序的效率问题和因 cut的出现而带来的有争议的语义问题. Gödel是 Prolog的进一步发展,其主要内容包括类型系统、控制机制、模块系统、元程序设计和输入/输出等部分.本文从这几个方面对 Gödel语言进行功能分析,重点讨论了 Gödel与 Prolog的区别,从中可以看到 Gödel的一些好的设计思想和理念,有助于新型程序语言的研究、设计和开发[1].

1 类型系统的分析与比较

在程序设计语言中,引入类型系统是非常有必要的.引入类型系统给所有的常数、变量、函数等对象加上类型信息,规定对象的取值范围和允许施加在其上的运算操作.这不仅便于知识表示,提高程序

收稿日期: 2008-12-16.
通信作者:昌 杰. E-mail: changjie_2003@163.com.

的可读性,便于程序员设计程序与开展合作交流;而且编译器也可根据类型系统说明的信息产生效率更高的代码,同时有助于捕获程序设计中的错误,大大增加了程序设计的正确性[2].

Prolog以 Horn子句逻辑为基础,是一种无类型程序设计语言.这种特点虽然简化了它的语言、编译和执行过程,但却大大降低了可表达性能力,使程序员设计的程序不易阅读和理解.在 Gödel语言中,由于引入了类型,通常语言说明以关键字 BASE、CONSTRUCTOR、CONSTANT、FUNCTION、PROPOSITION等开头,这些说明定义了程序中出现的各种数据类型和符号,有助于程序的阅读和理解.

例 1 Gödel语言的类型说明示例

```
BASE          Day, Person                          % 基类说明
CONSTRUCTOR   List/1.                              % 类型构造算子
CONSTANT      Nil: List(a).                        % 常量说明
              Monday, Tuesday: Day.
              Fred, Bill: Person.
FUNCTION      Cons: a * List(a) → List(a).         % 函数说明
PREDICATE     Append: List(a) * List(a) * List(a). % 谓词说明
```

如例 1所示,程序中的语言说明可以由以下几部分组成:

1)以关键字 BASE开头的语言说明给出了多态多类语言程序的基本类型,称为基类.

2) CONSTRUCTOR说明了用户命名的含有 n元类型参数的结构类型(称为构造子),可以从一些类型构造产生新的类型.

3) CONSTANT说明定义了常量.

4) FUNCTION和 PREDICATE分别说明了函数和谓词类型,对应 Gödel语言中的项和原子. Gödel语言的函数和谓词都是基于类型的,函数和谓词的每一个参数都需要进行类型说明.

在上面的示例中,Day和 Person是基类,CONSTANT说明 Nil是 List(a)类型的一个常量,Monday、Tuesday分别是 Day类型的常量,Fred、Bill分别是 Person类型的常量. FUNCTION说明了 Cons是一个二元函数,它把二元组(第 1个参数是 a类型,第 2个参数是 List(a)类型)映射到一个 List(a)类型的元素. PREDICATE说明了 Append是一个三元谓词,每个参数都是 List *(a)* 类型.

一元构造子 List本身不是一个类型,但程序中出现的所有基类和构造子都必须说明.没有构造子,则所有类型的集合只是所有基类的集合.如果有一个构造子被说明了,则类型的集合将把基类视为"常量",把构造子视为"函数"来构造获得函数类型,这样就可以构造诸如 List(Day)、List(List(Day))等可数无限的类型的集合.

例中, a是一个类型变量,允许反映 Gödel语言的参数多态性,它可以替换任意一个可以构造的类型.一个类型是一个项,它可以如下构造:将基类视为"常量",类型参数(选取子)视为"变量",构造子视为"函数".例如,对上述说明来说,所有类型的集合将是 Day、Person、List(Day)、List(List(Day))、a、List(a)等等.基于参数多态性,这里的谓词 Append能够对任何类型列表进行追加.

Gödel语言的类型系统是一个强类型系统,程序中的每个项和它的类型必须进行语言说明.尽管变量的类型说明不做强制要求,但为了保证其类型可由上下文推测出来,约定程序中有足够的信息可被编译程序和推理机用于判定类型并保证程序的安全执行,而不需要显式地给出变量的类型说明[2].

Gödel语言在引入类型后,显然大大提高了语言的可表达性能力,使其程序拥有更好的说明性语义.并且类型系统的引入给 Gödel语言的编译带来了方便,提高了执行效率,也更容易实现程序的并行执行[3].

2 控制机制的区别

控制机制主要针对语言的过程性语义.在 Prolog语言中,针对其 Horn子句,采用 SLD反驳—消解法来推导出最后的结果.这种方法无论从实际的操作过程还是从理论的推导上都是严谨正确的,它也是 Prolog语言说明性语义和过程性语义的桥梁.

例 2 设有 Prolog程序片段

1) Reverse([ ], [ ]).

2) Reverse([*H* | *T*], Rev) ← Reverse(*T*, *TR*) ∧Append(*TR*, *[H]*, Rev).

这里,Append(*X*, *Y*, *Z*)谓词将列表 *X*和 *Y*按顺序合并成新表 *Z*.如该程序目标子句为 Append([1, 2], [3, 4], *Z*),则 *Z*的输出结果为[1, 2, 3, 4]. Reverse(*X*, *Y*)谓词是将 *X*列表逆转得到新表 *Y*.如果目标子句为 Reverse([2, 4, 3], *Y*),容易得到 *Y* = [3, 4, 2].但对于目标子句 Reverse(*Y*, [2, 4, 3]),系统会因为回溯的无法停止而陷入无限循环.

Prolog引入了附加的控制语句 cut(!)以及重新调整定义子句和子目标的次序等方法来试图解决此

类问题.其中 cut最重要的作用是,丢掉在它所出现的规则中从头部开始直到“!”之间所有子目标由于成功而在数据库中作出的标记.使这些目标的再满足无法进行,从而防止回溯.它可以解决一些由于 Prolog最左文字优先规则而引起的的无限循环.例如:

例 3 设有 Prolog程序片段

1) member(*X*, *[X | T]*) ← *!*.

2) member(*X*, *[Y | T]*) ← member(*X*, *T*) .

member(*X*, *Y*) 谓词判断 *X*是否为列表 *Y*中的元素.在没有加“!”之前,执行目标子句 member(*X*, *[*1, 2, 3*]*)时, *X*将在分别输出 1, 2, 3后陷入死循环.引入“*!*”以后, *X*将在输出 1以后停止执行.但是,若要求输出所有解,“!”在任何位置的引入都无法解决此问题,即使改变子句和子目标的次序也无济于事,同时该方法缺乏通用性.

针对 Prolog语言控制机制存在的问题, Gödel语言对其控制机制进行了改进,引入 DELAY延迟控制剪枝操作.程序员可以通过适当的 DELAY延迟保证一个调用在被足够实例化后再推进,这样就避免了无法预料的失败,同时有助于程序员控制程序运行[2].

例 4 对应于同样的问题, Gödel语言采用的源程序如下:

```
PREDICATE    Reverse: List( a ) * List( a ).
DELAY        Reverse( X, Y ) UNTIL
NONVAR( X ) ∨NONVAR( Y ).

1) Reverse( [ ], [ ]).

2) Reverse( [ H | T ], Rev ) ← Reverse( T, TR )
∧Append( TR, [H ], Rev ).
```

其中,DELAY延迟的语法形式为 DELAY Atom UNTIL Cond, Atom为原子, Cond为条件集.这里, Cond为 NONVAR(*X*) ∧NONVAR(*Y*).程序表示在操作中只有当 Reverse( )内 *X*、*Y*中任意一项被实例化,才能对 Reverse( )所对应的子句进行处理,否则,延迟将挂起对谓词 Reverse( )的调用,直到条件被满足.

所以,当例 2中同样的目标子句 Reverse(*Y*, [2, 4, 3])被执行时,首先,目标子句满足 DELAY条件,在与子句 1匹配失败后,与子句二合一,生成 2个子目标 Reverse(*T*, *TR*)和 Append(*TR*, *[H]*, Rev).由于子目标 1不满足延迟条件被挂起,子目标 2被执行, Append( )谓词将 *TR*和 *H*分别实例化为 [2, 4]和 3.此时,子目标 1延迟条件满足,立刻将其执行唤醒,对 Reverse(*T*, [2, 4])进行操作.如此循环操作直到所有合一操作结束,将得 *Y*值为 [3, 4, 2].

从例子可以看出,延迟不仅解决了因 Prolog控制机制带来的无限制循环问题,而且还引入了并发执行,可提高系统执行的效率.

Gödel的剪枝操作 commit类似于 Prolog中的 cut机制,但它是建立在并发语言的 commit机制之上,具有更好的说明性语义.然而, Gödel的剪枝操作也会带来和 prolog中 cut一样的问题,如剪去了有效答案所在的分支等,但 Gödel语言通过增强对谓词的 DELAY说明就能够很好地解决这个问题.

对于上述程序,若没有对其增加 DELAY说明,当执行目标语句 member(*X*, [1, 2, 3]) ∧*X* =2时,第 1个子目标在与子句 1合一后得到 *X* =1结束回溯.这个结果与子目标 2相矛盾,从而无法得出正确的结果.当对其增加如下说明:

DELAY member(*X*, *[Y* | _]) UNTIL NONVAR(*X*) ∧NONVAR(*Y*).

对该目标子句,第 1个子目标由于不满足条件而被挂起,直到第 2个子目标执行将 *X*实例化为 2,立即将子目标 1唤醒.得到判断结果 TRUE,表示 *X*是列表中元素.

尽管剪枝会破坏程序的可读性和正确性,但合理的剪枝可以大大提高程序执行的效率,因此,剪枝在逻辑程序设计语言中仍具有重要意义.在 Gödel语言中,如果配合以强 DELAY说明,更将大大减少剪枝带来的负面作用,从而得到更加广泛的应用.

3　模块化程序设计的作用

模块化程序设计是大型软件系统开发的有效方法和技术之一. Prolog的设计没有引入模块化的设计方案,语言本身过于简单也使得它并不适用于大型软件的设计与开发.随着计算机应用领域的不断深化,对软件的效率、可靠性、易维护的要求越来越高,非模块化设计功能的 Prolog很明显不符合现代软件工程的基本要求.

Gödel语言的定位考虑了大型智能软件的开发,引入了模块系统.每个模块相当于一个构件,程序员将各个不同的构件进行组装设计可形成更大的程序,而每个构件尽量保持各自的独立性.模块的执行细节封装在模块内部,对其他构件保持透明[4].这样的设计使 Gödel语言融合了现代软件工程的思

想方法,更易于反映现代软件的设计理念.

例 5 Gödel语言模块系统的构成实例:

```
EXPORT      M5.
IMPORT      Lists
BASE        Day, Person.
CONSTANT    Monday, Tuesday, Wednesday, Thursday,
            Friday, Saturday, Sunday: Day;
PREDICATE   Append: List(a) * List(a) *
            List(a); Append3: List(a) * List(a) *
            List(a) * List(a).
LOCAL       M5.
            Append(Nil, X, X). Append(Cons(U,
            X), Y, Cons(U, Z)) ← Append(X,
            Y, Z).
            Append3(X, Y, Z, U) ← Append(X, Y,
            W) & Append(W, Z, U).
MODULE      M6.
IMPORT      M5.
PREDICATE   Member2: a * a * List(a).
            Member2(X, Y, Z) ← Append3(_, [X
            | _], [Y | _], Z).
```

模块 M5包含一个输出部分 EXPORT和本地部分 LOCAL,其中 BASE、CONSTANT和 PREDICATE已在类型系统中介绍,这里重点讨论模块与模块之间的连接以及内部的关系. 在模块 M5中,引入了 Lists模块,它是处理表结构的系统模块,通过这个模块,可以执行谓词 Append3,它的参数类型是表类型. M6模块引入了前面的自定义模块 M5,它除了继承 Lists模块以外,还使用了 M5模块中的 Append3来处理 Member2谓词. 其中,Append谓词用于将第 1个参数表和第 2个参数表合并成新表,并且第 1个表在第 2个之前. Append3定义了 3个表的合并操作. 在模块 M6定义的谓词 Member2中调用了 M5中 Append3. 当且仅当 X和 Y是列表 Z的元素且 X在 Y的前面 Member2为真. 如当询问 Member2(Friday, Monday, [Thursday, Friday, Monday])时,该程序将返回真.

同样的程序 M6,若用 Prolog编写,则除了编写谓词 member2(),还必须重新编写谓词 Append3(),增加了程序语言编写的重复性,结构上也不如 Gödel语言写的程序.

随着应用的不断深化,各类软件日趋复杂,软件生命周期中对开发效率、可靠性、易维护性、易管理等方面提出了更高的要求. 通过引入模块化系统,Gödel语言在描述计算时不仅结构清晰,可读性好,适合开发大型软件系统;而且通过引入类型系统实现了可支持抽象数据类型程序设计,为支持面向对象程序设计奠定了基础.

4 元程序设计及其应用

所谓元程序是指那些将其他程序包括自身作为数据进行处理,无需考虑其源程序编程语言特性的程序. 元程序的重要性主要表现在其大范围的应用上,如编译器、解释器、程序分析器和程序转换器等. 另外,当逻辑程序被用于人工智能时通常需要形式化的知识表示,这些知识可用逻辑程序来表示,元程序可被视为元推理器,根据这些知识进行推断[5].

元程序中如何表示目标程序是元程序中最重要的问题. 元程序的表示方法主要有 2种:基本表示和非基本表示. 前者将任一表达式表示为基本表达式,后者除了将变量用相应的元级变量表示以外,非变量标识符在元程序中表示为对应的常量和函数. Prolog采用非基本表示方法.

例 6 设有 Prolog程序:

1) member(X, [X | T]).

2) member(X, [H | T]) ← member(X, T).

若用 Prolog元程序之一的 solve标准解释器可表示为

1) solve(member(X, [X | T]).

2) solve(member(X, [H | T])) ← clause(member(X, [H | T]), member(X, T)) solve(member(X, T)).

当执行 solve(X, [1, 2, 3])时,可以分别得到 X = 1, 2, 3. 但是,这种非基本表示会在执行过程中由于元级变量的绑定带来一些非说明性语义问题. 例如,当目标子句为 Var(X) member1(X, [1, 2, 3]),其中元程序子句 Var(X)用来判断 X是否是自由变量,可以分别得到结果 X = 1, 2, 3. 但是,在执行目标子句 member1(X, [1, 2, 3]) Var(X)时,却由于 X被实例化为 1,而使得 Var(X)执行失败.

为了解决此问题,Gödel采用元程序的另一种表示方法,即基本表示. 基本表示的主要特点是将目标语言表示为元语言中的基本项. 上述程序可表示为

1) if (Atom(member, [var(0), Term(cons, [var(0), var(1)])]), true).

2) if (Atom(member, [var(0), Term(cons, [var(2), var(1)])]), Atom(member, [var(0),

var(1)]))

其中:语句 $H \leftarrow B$ 表示为 if (*H*, *B*),谓词 member(*X*, *Y*)表示为 Atom (member, [*X*, *Y*]),表项 [*X* | *Y*]表示为 Term (cons, [*X*, *Y*]),并且 var(0)、var(1)、var(2)分别代表上述元程序中的 3个变量 *X*、*T*、*H*. 在对于目标子句 member1 (*X* , [1, 2, 3]) Var(*X*)的执行过程中,由于基本表示不会对元变量进行值的绑定,因而不会因为子目标顺序的颠倒而导致不同的结果.

在动态元程序方面,元程序的设计主要有 3种方法:使用预定义模块、更新原程序和转换合成元程序[6]. 尽管 Prolog采用了元程序的设计方法,但由于其没有采用模块化设计且不提供增加删除元语句的功能. 所以, Prolog在结构上并不支持动态的元程序设计,而仅通过子句中参数值的改变而影响程序的控制流程,达到元程序设计的目的. 为了解决此问题, Prolog引入 cut对 Prolog运行机制进行控制,试图实现动态元程序设计的目标,但该语句并未从根本上改变程序本身,所以仍然具有很大的局限性.

尽管 Prolog语言在后期引入了若干语句可以对元程序数据库进行修改,例如 retract() 删除语句. 但 retract() 语句修改了正在执行的程序,而且删除效果在回溯过程中是不可逆的,很明显它也不具有说明性的语义.

Gödel语言为了满足动态元程序设计的要求,除了采用模块化程序设计,还提供了一系列添加删除程序的元语言子句.

例 7　设有 Gödel程序:

```
MODULE      M1.
IMPORT      Lists, Programs
PREDICATE   Appenda: List( a ) * List( a ) *
            List( a ).
            Appenda( X, Y, Z ) ←
            Append(X, Y, Z) ∧
            DeleteStatement(P, Lists, Append, P)
```

模块 M1引入了 Lists和 Programs模块, Lists模块中包括了谓词 Append (*X*, *Y*, *Z*),表示将列表元素 *X* 和 *Y* 合并成新表 *Z*. 例如,目标子句为 Append([1, 2], [3, 4], *Z*),可得 *Z* = [1, 2, 3, 4]. Programs模块提供了动态元程序设计所需要的谓词 DeleteStatement(*P*, *S*, *F*, *Q*),其中: *P*、*Q* 分别为一个程序的表示, *S* 是这个程序中一个打开的用户模块的名字, *F* 是出现在这个模块中的一条语句的表示. 该谓词表示将 *P* 程序引入的 *S* 模块中的 *F* 语句删除,接着将整个修改后的程序保存为 *Q*. 显然, DeleteStatement (*P*, *S*, *F*, *Q*)的引入,可视为改变 Gödel语言程序控制流程的一种元级控制策略. 并且配合模块化程序设计技术,具有改变源程序的能力,并仍然保持了良好的说明性语义.

对于目标 Appenda([1, 2], [3, 4], *Z*),返回结果 *Z* = [1, 2, 3, 4],同时,该目标的执行还将程序 *P* 中的 Lists模块的 Append语句删除,修改后的程序仍表示为 *P*. 这样使得模块 Lists中不含有谓词 Append,即别的用户无法再进行 Append调用.

这种处理元程序设计的方法除了采用基本表示外,还将修改后的程序另外保存,所以并没有破坏程序的说明性语义. 以后,配合多态多类的变量设计,这种元程序设计技术在实现各模块 (构件)重用和程序安全方面将发挥很大的作用,并在软件开发中融入和体现软件重用的思想.

尽管基本表示的表示结果复杂,元程序设计需要更多的程序代码,并且执行效率相对较低,但却能表达更复杂的程序设计思想,程序设计更符合人的思维方式. 而 Gödel语言支持抽象数据类型机制,并在其基础上提供了大量的操作,大大提高了程序的运行效率,可弥补了其基本表示的不足. 显然,在元程序设计方面, Gödel语言比 Prolog取得了更大的进步.

5　输入/输出部分

任何程序设计语言都需要考虑输入/输出部分,逻辑程序设计语言也一样. 在目前已经出现的逻辑程序设计语言系统中,输入/输出语句并不是语言本身固有的语言成分,而是根据用户的需要在系统的实现过程中添加的. Prolog语言中,对于输入/输出的设计一般是增加一些专门用于输入/输出的预定义专用谓词,这种方法简单、易行. 显然, Gödel语言也可以遵循这样一种方式解决输入/输出问题. 然而,还应该看到,由于引入了模块化技术, Gödel语言可以采用模块化设计方法,将输入/输出谓词限定在一个最小的程序片段中,即对各种输入/输出数据通过专门的输入/输出模块进行处理,可以更好地解决输入/输出问题.

可见,引入了模块化系统以后,输入/输出被当作一个单独的模块被调用和处理,尽可能降低输入/输出模块在整个说明性程序中所占的比例,大大提高了 Gödel语言的说明性语义,使得程序的执行更

简单、直观.

6 结 论

Prolog是当今应用最为广泛的人工智能语言.然而,它的无类型、较弱的说明性语义以及不支持动态元程序设计限制了它的继续发展和深入应用.之后,在其基础上发展的一个基于多态多类一阶逻辑的程序设计语言 Gödel,融合了模块化程序设计、抽象数据类型、元程序设计、延迟计算和剪枝操作等程序设计技术,具有以下特点:

1)采用基本表示和强类型系统,具有更好的说明性语义[7];

2)支持模块化程序设计和抽象数据类型,使得大型软件系统的实现成为可能;

3)动态元程序设计让程序的执行更加灵活多变,可支持许多更精巧的程序设计,表达一些更复杂的程序设计思想;

4)延迟计算和剪枝操作的引入,完善了程序执行控制机制,既提高了执行效率,也使得程序的输出结果更加精确,并保持程序的说明性语义.

随着这些技术的进一步应用和完善,丰富了逻辑程序设计语言和逻辑程序设计的内涵[8],使得 Gödel语言的应用范围更加广泛,程序执行的效率和灵活性也大大提高.相信随着研究的不断深入,Gödel语言将会对说明性逻辑程序设计产生深远的影响.

参考文献:

[1]刘椿年,曹德和. PROLOG语言,它的应用与实现[M].北京:科学出版社,1990: 64-94.

[2] HILL P M, LLOYD J W. The Gödel programming language [M]. Massachusetts: MIT Press, 1994: 3-23.

[3]王炳波,赵致琢,晏 松. Gödel语言类型系统[J].计算机工程与设计,2005, 26(12): 3432-3435.
WANG Bingbo, ZHAO Zhizhuo, YAN Song Type system in programming language Gödel[J]. Computer Engineering and Design, 2005, 26(12): 3432-3435.

[4]李松斌,赵致琢,李慧琪. Gödel语言对现代软件工程方法的支持[J].计算机时代,2006 (11): 1-3.
LI Songbin, ZHAO Zhizhuo, LI Huiqi The support of Gödel language for modern software engineering methods[J]. Computer Era, 2006(11): 1-3.

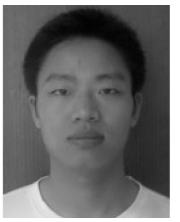
[5]王啸澜,赵致琢,李慧琪. Prolog语言与 Gödel语言中元程序设计方法的研究[J].厦门大学学报:自然科学版,2005, 44(6): 247-250.
WANG Xiaolan, ZHAO Zhizhuo, LI Huiqi Research on Meta-programming in Gödel language and Prolog lnguage [J]. Journal of Xiamen University: Natural Science, 2005, 44(6): 247-250.

[6] HILL P M, LLOYD J W. Analysis of meta-programs[C]// Meta-programming in Logic Programming Massachusetts: MIT Press, 1989: 23-52.

[7] GUNTER C A. Semantics of programming languages: structures and techniques, foundation of computer[M]. Massachusetts: MIT Press, 1992: 20-40.

[8] MITCHELL J C. Concepts in programming languages[M]. UK: Cambridge Univ Press, 2003: 4-15.

作者简介:



昌 杰,1983年生,男,硕士研究生,主要研究方向为逻辑程序设计语言 Gödel及其程序设计环境.

赵致琢,男,1957年生,教授,硕士生导师,主要研究方向为计算模型与分布式基础算法、软件开发方法学、计算机科学教育研究.先后获得 2000年福建省优秀教学成果奖一等奖、2001年国家级优秀教学成果奖二等奖.发表学术论文多篇,出版专著 2部.

李慧琪,女,1973年生,博士研究生,主要研究方向为逻辑程序设计语言 Gödel及其程序设计环境.