Journal of Southeast University (English Edition) Vol. 25, No. 1, pp. 31 – 35 Mar. 2009 ISSN 1003—7985

Shrek: a dynamic object-oriented programming language

Cao Jing[1,3] Xu Baowen[2,3] Zhou Yuming[2,3]

([1]School of Computer Science and Engineering, Southeast University, Nanjing 210096, China)
([2]Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)
([3]Software Quality Institute of Jiangsu Province, Nanjing 210096, China)

Abstract: From a perspective of theoretical study, there are some faults in the models of the existing object-oriented programming languages. For example, C++ does not support metaclasses, the primitive types of Java and C# are not objects, etc. So, this paper designs a programming language, Shrek, which integrates many language features and constructions in a compact and consistent model. The Shrek language is a class-based purely object-oriented language. It has a dynamical strong type system, and adopts a single-inheritance mechanism with Mixin as its complement. It has a consistent class instantiation and inheritance structure and the ability of intercessive structural computational reflection, which enables it to support safe metaclass programming. It also supports multi-thread programming and automatic garbage collection, and enforces its expressive power by adopting a native method mechanism. The prototype system of the Shrek language is implemented and anticipated design goals are achieved.

Key words: dynamic typing; metaclass programming; computational reflection; native method; object-oriented programming language

Shrek is a programming language which we have designed and implemented. It uses a Smalltalk-like syntax. The design goals of Shrek are: 1) It should be a class-based purely object-oriented language satisfying six principles proposed by Kay[1-2], and be dynamically, strongly typed[3-5]; 2) It should be of single-inheritance with Mixin[6-7] as its complement, and have a consistent class instantiation and inheritance structure which ensures safe metaclass programming[8-12]; 3) It should support automatic garbage collection, multi-thread programming, native methods and intercessive computational reflection[13-18].

Fig. 1 shows the architecture of the Shrek program development environment. Object space is an object container which handles object allocation, accessing, and recycling. Other modules access objects only through the interfaces supplied by object space. The debugging tools kit is used to inspect the runtime information of active objects in object-space. It includes class inspector, method inspector, object space inspector, etc. The class information window shows all declared classes and their fields and methods by using debugging tools. The compiler reads in class declarations, generates classes and method objects and parses method source codes into bytecodes. Interpreter is the execution engine. It runs as a thread so that many interpreter instances can extract bytecodes from different method objects, and execute them concurrently. They share the same object space and are mutexed through critical sections. The console accepts input, calls the compiler to parse it, then starts an interpreter thread to run it (other interpreter threads may be created during its running), and finally reports the execution results.

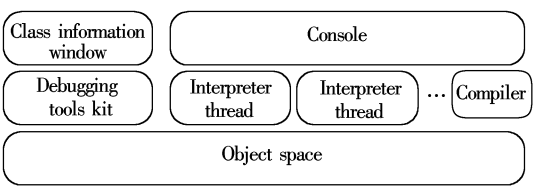

Fig. 1 The architecture of Shrek program development environment

The class library of Shrek currently includes thirty-four classes which can be classified as basic classes, numerical classes, container classes, character classes, input/output classes and thread classes.

1 Object Model and Dynamic Features

Fig. 2(a) shows the object model of Shrek. Excluding integer objects, an object is composed of two parts. One part is an array storing the value of the object. The array can be split into a head region and a field region. The hash value, object's size and its class pointer are stored in the head region separately. The stored size of an object excludes the size of head region. So its real size is the stored size plus three.

The other part is an entry of the object table, which is composed of the idle tag field, the reference counter field, the mark tag field, and the object handle field. When we want to allocate a new object, we must first find an idle table entry indicated by the idle tag. If there is no such entry, the new object cannot be allocated. If not, object-space allocates an array, saves its address in the object handle field, sets the idle tag as false, and returns the index of the entry. So the maximum number of active objects is decided by the size of the object table, which is 32 768 in the current version. A variable pointing to an object actually contains the index of the entry in the object table. The structures of the class object and the method object are shown in Fig. 2(b) (omit their object table entry).

The reference counter of an object records the number of other objects referring it at runtime. An object is recycled when its reference counter is zero, and its entry in the object table is also recycled (set idle tag as true). The reference counting algorithm can collect garbage immediately, but can-

Received 2008-07-15.
Biographies: Cao Jing(1980—), male, graduate; Xu Baowen(corresponding author), male, doctor, professor, bwxu@seu.edu.cn.
Foundation items: The National Science Fund for Distinguished Young Scholars (No. 60425206), the National Natural Science Foundation of China (No. 60633010), the Natural Science Foundation of Jiangsu Province(No. BK2006094).
Citation: Cao Jing, Xu Baowen, Zhou Yuming. Shrek: a dynamic object-oriented programming language[J]. Journal of Southeast University (English Edition), 2009, 25(1): 31 – 35.

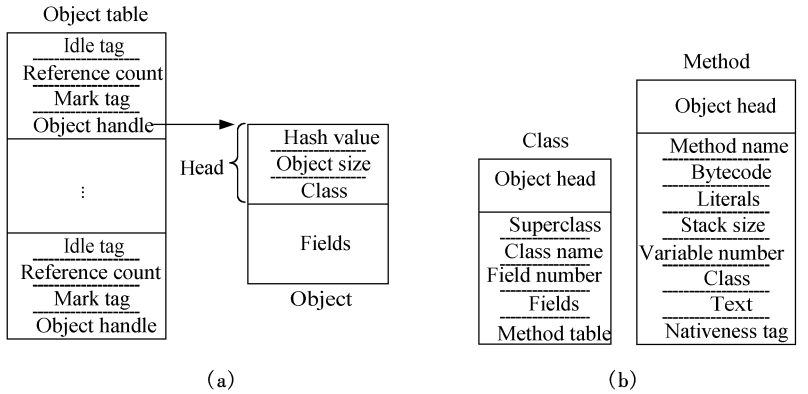

Fig. 2 Object model and structures. (a) Object model; (b) Structure of class and method

not reclaim cycle objects. Therefore, we also adopt a mark/sweep algorithm as a complement. Every object table entry has a mark tag field. When there is no idle entry, mark phase marks all active objects in object space through root objects, then sweep phase recycles unmarked objects.

Shrek has two primary dynamic features—dynamic typing and dynamic binding. The type of Shrek variable is resolved at runtime. Fig. 3 shows the structure of the value of a variable. The size of the value is 16 bits of which the lowest bit is a tag. If its value is zero, then the highest 15 bits contains an index of some object in the object table, through which we can acquire the object. Its class field contains the entry index of its class. We can acquire the class object alike, and then we know the type of the variable. If the value of the tag is one, the variable's type is Integer and the highest 15 bits contains the integer value based on complement encoding. So the current range of Integer is between −16 384 and 16 383.

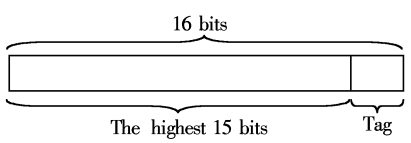

Fig. 3 The structure of a variable's value

All the Shrek methods are virtual, and method binding occurs runtime-wise as follows: 1) Acquire the method table of receiverClass(the class of the called object), look up the method whose name matches with the selector(the name of the message). If the method is found, return it. If not, go to the next step; 2) Check receiverClass. If receiverClass is Object, go to the next step. If not, acquire the superclass of receiverClass; assign the superclass to receiverClass, and go to 1); 3) Clear arguments(an array including arguments); store selector into it; let selector equal "message: notRecognizedWithArguments:", and go to 1).

From this, if the method lookup fails according to the original selector, the procedure will give up and try to find another method matching with "message: notRecognizedWithArguments:". The method is defined in Object which is the superclass of every other class. Therefore, it can be found definitely, and its function is to output the method lookup failure error.

2 Native Method and Structural Reflection

Some platform-dependent operations such as I/O operations, thread operations, etc. cannot be implemented as Shrek methods, but we can use a non-Shrek language(Java language) to implement them and call these native methods[19] in Shrek methods. Many methods in class IO and class SThread are declared as native methods, such as the following method which writes a byte to output the stream object.

```
@ writeByte: aByte
end
```

Native methods should be declared in Shrek class and implemented using Java. The declaration of a native method begins with "@" followed by a method signature. Because ":" is illegal in a Java method signature, the native method definition is the original name with ":" removed. For example, the definition signature of the above native method is "writeByte". Besides the arguments in the native method declaration, there are two more arguments which are object-space object and receiver in its definition. The simple implementation of the above method is shown as follows:

```
1 public Object writeByte( ObjectSpace os, Object receiver, Object anInteger) {
2    Object ioStream = os. basicAt( receiver, 3);
3    if( ioStream instanceof OutputStream) {
4       int value = os. integerValue( anInteger);
5       try{
6          (( OutputStream) ioStream). write( value);
7       }catch( Exception e) {
8          return os. getFalseObject( );
9       }
10       return os. getTrueObject( );
11    }else{
12       return os. getFalseObject( );
13    }
14}
```

Line 2 acquires the 3rd field of the receiver, which is an output stream object. Line 4 acquires the value which should be written out. Line 10 returns true. Line 12 returns false if the variable ioStream is not an output stream object.

The Shrek classes declaring native methods should dynamically load the Java classes defining these native methods in the initialization method as shown below:

```
initialize
   Shrek loadJavaClass: "IO". //load Java io. class
end
```

As described in Ref. [13], a reflective computational system is one in which implicit aspects of the system's structure and behavior are available for explicit inspection and manipulation. As described in Refs. [14 − 15], according to the reflection mode, there are two different models of reflection: behavioral reflection and structural reflection. Behavioral reflection is concerned with the reification of computations and their behavior. In contrast, structural reflection reifies the structural aspects of a program. According to reflection capability, there are two kinds of reflection: introspective reflection and intercessive reflection. Introspective reflection supports obtaining information at runtime about elements of the program under execution, without changing the program. For instance, in Java one can know at runtime the class of an object which one did not know before[19]. But Java does not have an intercessive reflection capability. Intercessive reflection supports changing the program during execution. For instance, Smalltalk allows the programmer to create a code that inserts a new method into a class at runtime[2].

Shrek supports intercessive structural reflection so that it allows the following operations at runtime:

- Acquiring an object's class, just as the statement "aClass = anObject class." does.
- Acquiring class's fields, just as the statement "fields = aClass fields." does.
- Acquiring class's methods, for example, the statement "aMethod = aClass methodNamed: #print." gets method "print".
- Creating class dynamically, for example, the statement "Object addSubclass: #Point fields: "*x y*"." creates class Point whose metaclass is Class and whose superclass is Object. Class Point contains *x* and *y*.
- When adding a new method in a class, for example, if one execute the statement "Point addMethod.", then one is waiting for entering a new method. After the method is entered, it will be parsed and added to the method table of Point.
- Modifying a method, for example, one can execute the statement "Point editMethod: #print." to change the function of method print.

Reflection simplifies Shrek and makes it easy to enhance Shrek by expanding its class library, without modifying its syntax. For example, class Mixin implemented by using reflection supplies Shrek with parameterized inheritance ability[16]. Aspect-oriented programming can also be implemented in Shrek by using reflection[17].

3 Metaclass Programming

In Shrek, there are two relations between classes. One is the inheritance relation between classes, and the other is the instantiation relation between a class and its metaclass[9]. Fig. 4 shows part of class instantiation and the inheritance structure of Shrek. As shown, the most important two classes are Object and Class. Object, the root of the inheritance tree, has no superclass and AbstractClass as its metaclass. Class, the root of the instantiation tree, has Object as its superclass and itself as its metaclass.

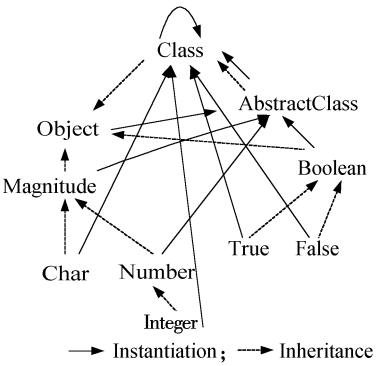

Fig. 4 Part of class instantiation and inheritance structure of Shrek

The function of class is to model the behavior of its instance. We can customize class's behavior through programming its metaclass. For example, Java supports abstract class which cannot instantiate concrete objects by using keyword "abstract". But Shrek can achieve this without modifying syntax. Classes which are the instances of AbstractClass play the role of abstract class. AbstractClass is the subclass of Class, and its definition is shown as follows:

```
class AbstractClass < Class
…
    new
       Shrek error: "Class " + self name toString + " cannot make an
Instance.".
          end
   …
end
```

It overrides method new, therefore if its instances such as Object are asked to create new instances, error message will be printed. As shown in Fig. 4, Object, Boolean, Magnitude and Number are instances of AbstractClass; Char, True, False and AbstractClass are instances of Class.

Programming with metaclass has two symmetrical kinds of compatibility issues caused by inter-lever communication[10]. They are: 1) Upward compatibility issue. As shown in Fig. 5(a), MetaA and MetaB are metaclasses of A and B, respectively. A defines method i-foo which calls method c-bar defined in MetaA. If B inherits A, but MetaB does not inherit MetaA, then B cannot respond to a c-bar message sent by its instances; 2) Downward compatibility issue. As shown in Fig. 5(b), MetaA defines method c-foo which calls method i-bar defined in A. If MetaB inherits MetaA, but B does not inherit A, then instances of B cannot respond

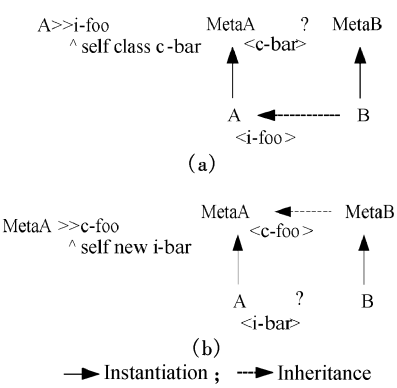

Fig. 5 Upward and downward compatibility

to a message i-bar sent by B.

There are two problems in existing metaclass compatibility guarding mechanisms. One is that they cannot guard upward and downward compatibility simultaneously, such as SOM[9,11] and NeoClasstalk[11]. The other is that they may be so strict that their expressiveness is weak, such as CLOS[11,18] and Smalltalk-80[2,11]. Shrek adopts non-strictly parallel single inheritance to guard compatibility[12]. As shown in Fig. 6, there are three inheritance ways, they are: class owns the same metaclass as its superclass(see Fig. 6(a)); the metaclass of a class inherits the metaclass of its father directly(see Fig. 6(b)); the metaclass of a class inherits the metaclass of its father indirectly(see Fig. 6(c)). The non-strictly parallel single inheritance is coherent for Shrek, and it is easy to prove that any class in the inheritance structure is compatible with its metaclass.

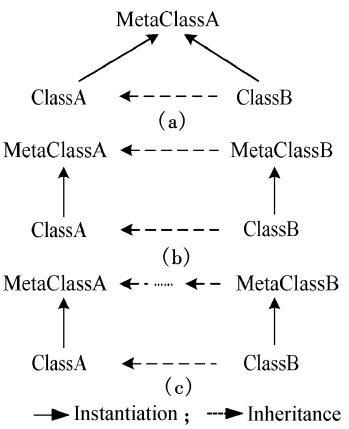

Fig. 6 Non-strictly parallel single inheritance of Shrek

4 Conclusion

Shrek is a purely object-oriented dynamically strongly typed programming language which supports safe metaclass programming, intercessive structural reflection, native methods, multi-thread programming, automatic garbage collection, etc. But recently, it has been found that Shrek has some shortcomings. For example, it does not support exception; its index of object table is only 15-bits; its class library is poor, etc. In the future, we are going to perfect Shrek language, increasing its feasibility and making it a platform for language research.

Appendix The syntax of Shrek

```
classDefinition: : =
    [comment] classHead [comment] [fieldsDefinition]
     [comment] {methodDefinition} * "end"
classHead: : = "class" [metaclassName: : ] className
    " < " className
fieldsDefinition: : = "fdef" [fieldName {", " fieldName}
    * ] "end"
methodDefinition: : = normalMethodDefinition |
    nativeMethodDefinition
normalMethodDefinition: : =
    messagePattern temporaryDeclaration methodBody
    "end"nativeMethodDefinition: : =
    "@ "(unaryMessagePattern|keywordMessagePattern)
    "end"
messagePattern: : = unaryMessagePattern
    | binaryMessagePattern | keywordMessagePattern
unaryMessagePattern: : = methodName
binaryMessagePattern: : = binarySelector parameterDeclara-
    tion
keywordMessagePattern: : =
    methodName": " parameterDeclaration {methodName
    ": " parameterDeclaration} *
temporaryDeclaration: : = " | " {temporaryName} * " | "
    methodBody: : = {[comment] statement} *
statement: : = ["^"] expression ". "
Expression: : = assignmentExpression | messageExpression
assignmentExpression: : = targetVariable " = " expression
targetVariable: : = fieldname | parameter | temporary
    | symbol
messageExpression: : = term message {"; " message} *
message: : = unaryMessage | binaryMessage
    | keywordMessage
unaryMessage: : = messageName
binaryMessage: : = binarySelector(term unaryMessage
    | "(" expression ")")
keywordMessage: : =
{messageName": "(term binaryMessage |
    "(" expression ")")}+
term: : = "(" expression ")" | block | primitive |
    variable | Literal
block: : = "[" {blockTemporaryDeclaration} * " | "
    {statement} * "]"
blockTemporaryDeclaration: : = ": "temporaryName
primitive: : = " < " number {variable} * " > "
variable: : = fieldname | parameter | temporaryName
    | globalVariableName
Literal: : = ArrayLiteral | IntegerLiteral | FloatingPoint-
    Literal | BooleanLiteral | CharacterLiteral |
    StringLiteral | SymbolsLiteral | NullLiteral
ArrayLiteral: : = "#(" {Literal} * ")"//for example,
    #("food" "taxes")
BooleanLiteral: : = "true" | "false"
CharacterLiteral: : = "' " ascii "' "//for example, ' a'
StringLiteral: : = """ text """//for example, "hello"
SymbolsLiteral: : = "#"Identifier//for example, #bill
NullLiteral: : = "nil"
Comment: : = singleLineComment | multiLineComment
singleLineComment: : = "//" text
multiLineComment: : = "/ * " text " * /
```

References

[1] Kay A C. The early history of smalltalk[J]. *ACM SIGPLAN Notices*, 1993, **28**(3): 69 − 95.

[2] Goldberg A, Robson D. *Smalltalk*-80: *the language and its implementation*[M]. Massachusetts: Addison Wesley, 1983.

[3] Scott M L. *Programming language pragmatics*[M]. 2nd ed. Morgan Kaufmann Press, 2005.

[4] Friedman D P, Haynes C T, Wand M. *Essentials of programming languages* [M]. 3rd ed. MIT Press, 2008.

[5] Meijer E, Drayton P. Static typing where possible, dynamic typing when needed: the end of the cold war between programming languages[C]//*OOPSLA Workshop on Revival of Dynamic Languages*. New York: ACM Press, 2004.

[6] Bracha G, Cook W. Mixin-based inheritance [C]//*Proc*

Joint ACM Conference on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming. New York: ACM Press, 1990: 303 - 311.

[7] Ancona D, Lagorio G, Zucca E. Jam—designing a Java extension with mixins[J]. *ACM Transactions on Programming Languages and Systems*, 2003, **25**(5): 641 - 712.

[8] Bouraqadi N. Safe metaclass composition using mixin-based inheritance[J]. *Journal of Computer Languages, Systems and Structures*, 2004, **30**(1/2): 49 - 61.

[9] Forman I R, Danforth S H. *Putting metaclasses to work* [M]. New York: Addison Wesley, 1998.

[10] Graube N. Metaclass compatibility [C]//*Proc of the* 4*th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York: ACM Press, 1989: 305 - 315.

[11] Bouraqadi N, Ledoux T, Rivard F. Safe metaclass programming[C]//*Proc of the* 13*th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York: ACM Press, 1998: 84 - 96.

[12] Cao Jing, Xu Baowen. Safe metaclass programming based on non-strictly parallel single inheritance[C]//*Proc of the* 5*th Conference on the Development and Education of Programming Languages*. Beijing: Tsinghua University Press, 2006: 19 - 26. (in Chinese)

[13] Smith B C. Reflection and semantics in Lisp[C]//*Proc of the* 11*th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York: ACM Press, 1984: 23 - 35.

[14] Maes P. Concepts and experiments in computational reflection[C]//*Proc of the* 2*nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York: ACM Press, 1987: 147 - 155.

[15] Ferber J. Computational reflection in class based object oriented languages[C]//*Proc of the* 4*th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York: ACM Press, 1989: 317 - 326.

[16] Sullivan G. Aspect-oriented programming using reflection and metaobject protocols[J]. *Communications of the ACM*, 2001, **44**(10): 95 - 97.

[17] Kojarski S, Lorenz D H. AOP as a first class reflective mechanism[C]//*Proc of the* 19*th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York: ACM Press, 2004: 216 - 217.

[18] Kiczales G, Rivieres J, Bobrow D G. *The art of the metaobject protocol*[M]. MIT Press, 1991.

[19] Gosling J, Joy B, Steele G, et al. *The Java language specification* [M]. 3rd ed. New York: Addison Wesley, 2005.

Shrek:一个动态面向对象程序设计语言

曹 璟[1,3] 徐宝文[2,3] 周毓明[2,3]

([1] 东南大学计算机科学与工程学院,南京 210096)

([2] 南京大学计算机科学与技术系,南京 210093)

([3] 江苏省软件质量研究所,南京 210096)

摘要:以理论研究的视角,现有面向对象程序设计语言的理论模型存在不足,如C++不支持元类,Java和C#的基本类型不是对象等.为此,设计了一种程序设计语言Shrek,将多种语言特性和语言设施统一在一个简洁、一致的模型下实现.Shrek语言是基于类的完全面向对象语言,拥有动态强类型系统,采用了与Mixin相结合的单继承机制.该语言具有协调一致的类对象结构,具备结构化计算反射能力,能够进行安全的元类程序设计.另外,它还支持多线程程序设计和自动垃圾回收,并通过本地方法机制极大地增强了自身的表达能力.该语言的原型系统已经实现,达到了预期的设计目标.

关键词:动态类型;元类程序设计;计算反射;本地方法;面向对象程序设计语言

中图分类号:TP312