MOOL: an Object-Oriented Language with Generics and Modules

Maria Lucia Barron Estrada[1], Ramón Zatarain Cabada[1], and Ryan Stansifer[2]

[1]Instituto Tecnológico de Culiacán, Av. Juan de Dios Bátiz s/n, Col. Guadalupe, Culiacán, Sin. CP 80220 México
mbarron@fit.edu
[2]Florida Institute of Technology, 150 W. University Blvd. Melbourne, FL. 30901 USA
ryan@cs.fit.edu

Abstract. Though most developers are concerned with only a few languages, programming languages continue to evolve. Researchers continue to develop new features and experiment with new combinations of features in order to design languages that are easier to program in. In this paper we describe the programming language called MOOL (Modular Object-Oriented language). MOOL is a simple class-based object-oriented language that supports generic programming by parameterized classes and interfaces [4]. It contains an independent module mechanism to allow the development of large programs in a safe manner. It is this combination of features that differentiates MOOL from C++, C#, and Java. A prototype compiler is under development that will translate MOOL to both the MSIL for .NET platform and bytecode for Java.

1 Introduction

There are many language features that programmers expect to find in modern programming languages. Every programming language provides some of them, usually in different combinations and variations. It is important to begin with a rough list of these features.

•Control constructs	
•Concurrency	
•Input/output	
•Aggregation	for structuring large programs
•Generics/templates	for convenient code-reuse
•Inheritance	for convenient code-reuse

We are not concerned with the first three in this paper. Rather we focus on the last three. Our Goal is to design an imperative, type-safe, class-based language with aggregation, generics, and inheritance. We want a language that is as simple and natural as possible.

We look at several of the major languages and how they approach aggregation, generics, and inheritance.

Ada [1] and SML [8] both have a well-defined module mechanism. Both support generic programming. Support for inheritance was added later to Ada.

Java [3] supports inheritance and classes and it is planned to add support for generics programming. The situation for C# [2] is similar.

Both Java and C# relegate aggregation to a minor role.

Only Modula-3 [9] was designed with all three (aggregation, generics, and inheritance) in mind from the beginning.

With respect to our goal all these language fall short.

Ada and Modula-3 are deficient (from our point of view) because objects/classes are subordinate.

SML is inspirational in many ways, in particular with universal polymorphism type reconstruction, and modules, but it is not a traditional, imperative language. And it does not have classes, though its cousin OCAML [10] does.

In Java, C#, C++ [11] modules play a reduced role. In these languages the class construct is overburdened.

Classes play many roles in languages that do no support another mechanism to structure programs [12]. The separation of classes and modules in two distinct elements allows using them independently; it also permits to use them to generate applications without forcing the developer to use a specific paradigm. However, separate classes en modules in two distinct constructs is not an easy task. Some language designers have accomplish this task but not without sacrificing other elements, i.e. MOBY [7] and OCAML [10]. These two languages are descendents of SML and they incorporate classes into the language carrying on all the elements previously defined in it.

The absence of parametric polymorphism in object-oriented languages like Java and C# has annoyed developers to some extent that both languages are in the process to get a mechanism to support parametric polymorphism. MOOL allows the definition of parameterized classes, class interfaces, methods and functions. Parameterized types and functions allow developers to abstract over types. This will enhance code reuse in a safe manner.

MOOL provides different constructs for classes and modules. Modules are containers, which are used to generate programs or code fragments. Classes on the other hand, are useful to create programs using the object-oriented style. Thus both imperative and object-oriented programs can be design without forcing the developer to use any style. A complete definition of the language can be found at [4]. The grammar of MOOL is presented in BNF notation as an appendix in [4].

2 MOOL

MOOL - Modular Object-Oriented Language - is a simple, general-purpose, statically typed, class-based, object-oriented programming language. It provides a module construct with interface and implementation separated to create large programs, class construct to define and generate objects, and supports the definition of generic code using parameterized types.

MOOL provides two kinds of universal polymorphism. Parametric polymorphism is supported with parameterized types and type variables. Subtype polymorphism is provided to be able to use an object of a subtype where an object of its supertype is expected.

There is only one kind of type in MOOL: reference types. There is a hierarchy of types defined with object at the top. Everything is a reference to an object of certain type. Automatic boxing and unboxing is provided to use the values contained in certain objects.

MOOL includes classes, inheritance, polymorphism, dynamic dispatch, and late binding to support the object-oriented programming style. It also includes functions to create procedural programs that do not require the use of objects.

2.1 Definitions

Program. A MOOL program is a set of compilation units. A compilation unit is either a *module interface* or a *module implementation.* A program specifies a sequence of statements to be executed in some order.
Identifiers. Identifiers are names used to define and refer to some elements in a program such as variables, functions, types, etc.
Expression. An expression is a construct in the language in which a combination of operators and operands specify a computation that produces a value.
Predefined types hold numeric or boolean values. They are integer, float, and boolean. There is also a special reference value called **null**. The user can create other reference types using classes, class interfaces, functions and arrays.

2.2 Module Interface and Implementation

A module is the basic unit to create a simple program or to create a code fragment that can be combined with other modules to create larger programs. Functions, classes and class interfaces are part of modules and cannot be defined independently.

Modules are static units to encapsulate elements, hide information and separate compilation. Modules contain two parts: a **module interface** that describes the signature of the module and the **module implementation** that contains the implementation of the signature. Modules define the namespace structure to refer to qualified names. They define two scopes: internal and external.

In this section we present the two parts of the module construct.

Module Interface

A module interface is a specification of the services a given module provides to others. A module interfaces reveals the public parts of a module. Information hiding can be achieved by restricting the interface to contain only a subset of the elements defined in the module implementation. By default all members of a module interface are public.

The language contains a module interface called *IMain*, which contains the *main* function. The *main* function receives an array of string elements and its result is void.

Any module may implement *IMain* providing code for the *main* function. The *main* function is the point where the program starts execution.

Example of the module interface *IMain* containing the definition of the *main* function.

```
module interface IMain {
  void main (String [] args);
}
```

Module Implementation

A module implementation contains the definition of all the elements shown in the module interface as well as some other elements that are only visible inside the module. A module implementation can contain constants, variables, types (functions, classes, and interfaces) and an initialization part (init Block), which is used to initialize the elements of the module before they are loaded to execution.

The elements declared inside a module are valid in the scope they are declared. All elements listed in the module interface are public elements unless they are annotated as protected. The module exports the interfaces listed in the *ModuleInterfaces* part. A module implementation has the form:

module Identifier ModuleInterfaces ModuleBlock

Example of of a module implementing the *IMain* module interface.

```
module Hello implements IMain {
  import System;
  void main (String [] args){
      IO.printLine(''Hello world!'');
  }
}
```

Scope

Modules define two scopes; internal and external. The combination of modules and classes provides control over class members' visibility. Listing a class interface in a module interface allows hiding some members of the class in the module implementation. A class declared in the module interface can annotate its members as protected. By default, all members are public. The combination of members that are listed or not listed in the module interface gave us several views of them in different scopes.

Example of a module interface and its implementation defining different views of elements.

```
module interface IM1 {
   class interface IC1 {
      void m1( );
      protected void m2( );
   }
   class C1 extends object implements IC1{
```

```
            constructors
                C1( );
        }
    }
    module M1 implements IM1 {
        class C1 {
            fields
              ...
            constructors C
                C1 ( ) { ...}
            methods
             void m1 {...}
             void m2 {...}
             void m3 {...}
           }
          // other elements of module M1
    }
```

The module interface *IM1* contains two declarations, a class interface and a class. In the class interface *IC1* two methods with different access (public and a protected) are declared. The module implementation *M1* contains the complete definition of class *C1*. Class *C1* contains three methods, but only two of them were listed in the class interface *IC1* in module interface *IM1*. All members of a class are visible inside the module and they are available for objects and derived classes. A protected member of the class listed in the class interface is visible outside the module only for derived classes.

The example above contains a definition of class *C1* with three methods: *m1*, *m2*, and *m3*. Tables 1 and 2 show how these members are available for users and derived classes inside and outside the module implementation.

Table 1. Visibility of methods of class C1 inside the module implementation

	Derived classes	Users
Member m1	visible	visible
Member m2	visible	visible
Member m3	visible	visible

Table 2. Visibility of methods of class C1 outside the module implementation

	Derived classes	Users
Member m1	visible	visible
Member m2	visible	non-visible
Member m3	non-visible	non-visible

2.3 Class Interface and Class Definition

MOOL provides single implementation inheritance and multiple interface inheritance. Classes are organized in a hierarchy with class object at the top. The class hierarchy is build with the definition of new classes and the specialization of existing ones. A class inherits from another class and implements one or more class interfaces. Classes that do not explicitly extend another class, implicitly inherit from object.

The class hierarchy does not organize the structure of a program; it is defined to allow code reuse and incremental definition of classes. The class mechanism is not used to support namespace management nor visibility control.

MOOL uses nominal subtyping, which means that classes define types and subclasses define subtypes.

Class Interface

A *class interface* is a type declaration that provides a specification rather than an implementation for its members. *Class interface* types are used to provide multiple inheritance in MOOL. Any class interface implemented by a class is a supertype of that class. A class interface declaration has the form:

class interface Identifier [TypeParameters][ExtendsInterfaces] InterfaceBodyDec

The class interface identifier must be unique in the module where it is defined. The identifier may be followed by an optional list of type parameters to declare a parameterized interface type which are presented in section 2.4.

Example of of a class interface definition.

```
class interface IFigure {
     void move (integer dx,dy);
     void draw();
}
```

Class Definition

MOOL contains a construct to define classes as extensible templates that encapsulate state and behavior. Classes in MOOL have three distinct roles: class definition, class specialization, and object creation. A class may inherit from another class and it may implement one or more class interfaces. A derived class can override an inherited method but it must be explicitly declared. It can also shadow some members but it must be explicitly declared to avoid unintentional shadowing of members.

Example of two class definitions using inheritance.

```
class Figure implements IFigure {
   fields
      Point center;
   constructors
      Figure () {center.x = 0; center.y =0;}
```

```
        Figure (integer x, integer y) {
                center.x = x; center.y =y;
        }
    methods
        void move (integer dx, integer dy) {…}
        void draw() {… }
}
class Circle extends Figure implements ICircle {
    fields
        integer ratio;
    constructors
        Circle () { this(0,0,0) }
        Circle (integer r) { this(0,0,r); }
        Circle (integer x, y, r) { super(x,y);
                                   this.ratio=r;}
    methods
        float area () { // implementation of area   }
        override void draw(){//new impl. of draw    }
}
```

A class declaration provides a class type that can be used to declare object variables of that type. Classes are used to generate objects dynamically. All objects created with a specific class have the same behavior at runtime and it cannot be modified. Objects are created applying the new operator to a class constructor. A class declaration has the form:

class Identifier [TypeParameters] [SuperClass] Interfaces ClassBodyDec

TypeParameters is an optional part that specifies that the class is generic. Generic classes are explained in detail in section 2.4. *SuperClass* is an optional part that specifies the direct superclass of the class. *Interfaces* specifies the list of interfaces that are implemented by the class. *ClassBodyDec* contains the declarations of the members of the class and the implementation of its constructors and methods. Classes have four kinds of members: class variables, fields, constructors, and methods.

Class variables. Class variables are special members that are shared by all instances of the class. They are allocated once for the lifetime of the program.

Fields. Fields are also called instance variables. Each object has a copy of the fields declared in the class. A field declaration can hide an inherited field if it has the same name and type but the declaration has to be preceded by the shadow access modifier.

Constructors. A constructor is a special function that has the same name as the class and does not specify a return type. A constructor is used in the creation of instances of the class. A class can contain many constructors with different signatures. Constructors must be part of a class declaration in a module interface if they are meant to be available for users or specializers.

Methods. Methods are functions defined inside a class that are always dynamically dispatched. They implement the behavior of objects. All methods of a class are available inside the module that contains the class definition. A class can contain two or more methods with the same name if their signatures are different. A method with the same name and signature than one inherited may override it, if it is annotated as **override**. A method can hide an inherited method with the same name and signature if it is annotated as **shadow** and not **override**. Both methods will be available using a complete qualified name. By default all methods can be overridden in subclasses.

Modifiers
Access modifier. There is one access modifier called **protected**. Any element of a class interface annotated as protected can be used in derived classes. Protected members are not available for clients.
Member modifier. There is one member modifier called **shadow**. A field or method of a class can be annotated as shadow if it has the same name as one inherited. It is used to hide the inherited member. Both members are available for access. The member defined in the parent class can be accessed using a fully qualified name, casting the object to its parent class, or using *super*. The new member can be accessed with the dot notation.
Method modifier. There is one method modifier called **override**. A method annotated as **override**, overrides an inherited method. The signature of the method must follow the subtyping rules defined in section 6.2.5.

Subtyping for classes and class interfaces. The subtyping relationship between classes is defined explicitly when a class is declared. A class that extends another class is a subtype of the extended class. If a class doesn't extend another class, it implicitly extends object. A class is also a subtype of any class interface it implements.

Derived classes must follow the subtype rule for functions when a method is overridden to ensure type safety.

MOOL allows changes of types in subclasses as follows: *invariant* – no type changes are allowed for fields, *covariant* changes are allowed for result types of functions, and *contravariant* changes for function arguments.

2.4 Generic Classes and Class Interface

Generics are abstractions over types. MOOL provides support for the definition of generic types, and type variables.
In this section we present the definition of type variables, and generic types with different kinds of constraints, and how they can be used to create instances of them.

Type variables
A type variable is an identifier with the same features as other identifiers but it stands for a type. Type variables are introduced in parameterized types to represent a

type parameter. They are defined after the identifier of the type declaration and they can be bounded to other types to constraint the type that can be used in instantiations.

Type constraints

Generic code can be defined for all the types available in the system is called *unconstrained genericity* or for some types that hold some properties which is called *constrained genericity*. A bound is declared using the **implements** keyword. Type parameters may contain recursive bounds as in the example shown next.

Example of a parameterized class with a recursively bound type paramcter.

```
class interface IOrderable <T> {
   integer compareTo (T elem);
}
class interface IOrderedList < T > {
      T remove();
      void insert (T elem);
}
class OrderList <T implements IOrderable <T>>
         implements IordeList<T> {
   // class implementation…
}
```

Generic types

In MOOL classes, class interfaces, and functions can be defined to be generic. A generic type contains a list of type parameters with specific bounds. The bounds of the type parameters restrict the types of the actual parameters when an instance of the generic type wants to be created.

Generic functions. A generic function is a function that has a list of type parameters. It is called in a similar way to that of a non-generic function, except for the type parameters. A generic function named *swap* that receives a type parameter called *T* and three formal parameters is shown next.

Example of of a generic function called *swap*.

```
void swap <T> ( T [] a, integer i, integer j) {
     T temp = a[i];
     a[i] = a[j];
     a[j] = temp
}
```

Generic classes. A generic class contains a list of type parameters enclosed in < >. The type parameters can be bounded to other types.

Example of of a generic class interface and its generic class.

```
class interface IList <T> {
   T head ( );
   void cons(T elem);
}
class List < T >  implements IList < T > {
   fields
      ...
   constructors
      List<T> () {...}
   methods
      T head ( ) { ...    }
      void cons (T elem) {...  }
   ...
}
```

Generic class interfaces. A generic class interface contains a list of type parameters enclosed in < >. The type parameters can be bounded to other types.

2.5 Declarations

A declaration introduces a name for a variable, a constant, a function, or a type that is valid in a scope delimited by the block that contains it. Repeated names for variables are not allowed in the same scope. There are four kinds of declarations: constants, variables, functions, and types.

2.6 Statements

Statements execute actions. They are used to control the flow of execution of a program. Some statements are simple and some others contain other statements as part of their structure. Statements in MOOL are very similar to those in Java, C# and other languages. In this section we present some of the statements supported in MOOL without describing them exhaustively due to their similarity with other languages..

Assignment

An assignment statement has the form LHS = RHS. It requires checking type compatibility between the expression at the LHS and the value generated by the expression at the RHS.

Function Call

A function call could be an expression or part of it if it returns a value.

Continue, return, and break

These statements are used to break the execution flow and continue the execution with the next statement.

Block

A block statement is delimited by curly brackets and may contain local variable declarations and a sequence of statements. It defines a scope where local variables declared inside are valid.

For

The for statement contains a controlling-loop part and a block.

While

The while statement is a conditional loop that executes the block while the velue of the espresion is *true*.

If

The if statement contains an expression and a body, delimited by curly brackets. The body contains a statement and an optional else part.

Switch

A switch statement contains an expression and a body. The body defines a set of cases for which specific actions are defined and a default clause.

3 Translation

We are implementing a compiler for the MOOL language. The compiler produces target code for the .NET and JVM platforms.

The compiler reads the input file (MOOL source code) and then, in one step, executes lexical, syntactic, and semantic analysis, and intermediate code generation (abstract syntax tree). Another step traverses the abstract tree, producing the target code (.net or bytecode) as the output. We also are implementing a preprocessor that takes a MOOL program as an input and produces C# or Java code as an output (target code). Both schemes allow us to test different types of programs using all the features of the source and target languages.

4 Conclusions

We have presented MOOL, which is a new general-purpose programming language where the roles of classes and modules are separated and generic programming is supported.

MOOL enables object-oriented programming defining hierarchies of classes with single implementation inheritance and multiple interface inheritance. MOOL enables also the implementation of large programs providing modules - static units of encapsulation, information hiding, and reuse - and module interfaces to describe their interconnection. Generic programming is sustained by parameterized types.

Our language is similar to other programming languages in many ways. We adopted a related Java and C# syntax which both descend from C. We can say that MOOL's module system is based on the module system of Modula-3 and the class mechanism is a simpler version of Java and C# classes.
MOOL is not an extension of any other language despite of the similarities whit other languages.

References

1. United States Department of Defense. *Reference manual for the Ada programming Language.* GPO 008-000-00354-8, 1980.
2. Standard ECMA-334. *C# Language Specification* [Online] http://www.ecma.ch December 2001.
3. Ken Arnold and James Gosling. *The Java^{TM} Programming Language.* Addison Wesley. 1998.
4. Barron-Estrada, M. L., *MOOL: an Object-Oriented Language with Generics and Modules.* Ph.D. Dissertation. Florida Institute of Technology. Melbourne, Florida, USA, May 2004.
5. Kim Bruce. Foundations of Object-Oriented Languages Types and Semantics. MIT-Press 2002.
6. Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. "On binary methods." In *Theory and Practice of Object Systems*, 1(3): 221-242, 1996.
7. Kathleen Fisher and John Reppy. Foundations for MOBY classes. Technical Memorandum, Bell Labs, Lucent Technologies, Murray Hill, NJ, February 1999.
8. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts. 1990.
9. Greg Nelson, editor. *Systems Programming with Modula-3.* Prentice-Hall, Englewood Cliffs, NJ, 1991.
10. Didier Rémy. *Using, Understanding, and Unraveling the OCaml Language.* In Gilles Barthe, editor, *Applied Semantics. Advanced Lectures. Volume 2395* of Lecture Notes in Computer Science, pages 413–537. Springer Verlag, 2002.
11. Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991
12. Clements Szypersky. Import is not inheritance; why we need both: modules and classes. In *Proceedings of ECOOP '92, European Conference on Object-Oriented Programming,* Utrecht, The Netherlands, June/July 1992. Volume 615 of Lecture Notes in Computer Science, pages 19-32, Springer-Verlag Berlin Heidelberg 1992.