MOOL: an Object-Oriented Programming Language with Generics and Modules.

by

María Lucía Barrón Estrada

Maestro en Ciencias, en Ciencias Computacionales
Instituto Tecnológico de Toluca
México

Licenciado en Informática
Instituto Tecnológico de Culiacán
México

A dissertation submitted to
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Science

Melbourne, Florida
May, 2004

We the undersigned committee hereby recommend
that the attached document be accepted as fulfilling
in part the requirements for the degree of
Doctor of Philosophy in Computer Science

"*MOOL:* an Object-Oriented Programming
Language with Generics and Modules,"
a dissertation by María Lucía Barrón Estrada

_____
Ryan Stansifer, Ph.D.
Associate Professor, Computer Sciences
Dissertation Advisor

_____
Phil Bernhard, Ph.D.
Associate Professor, Computer Sciences

_____
Pat Bond, Ph.D.
Associate Professor, Computer Sciences

_____
Dennis E. Jackson, Ph.D.
Associate Professor,

_____
William Shoaf, Ph.D.
Associate Professor and Department Head, Computer Sciences

Abstract

Title: *MOOL*: an Object-Oriented Programming Language with Generics and Modules.
Author: María Lucía Barrón Estrada
Major Advisor: Ryan Stansifer, Ph.D.

Modern object-oriented languages like Java and C# do not support parametric polymorphism and do not have a traditional module system to allow the development of large systems. They overload the class mechanism with several tasks and they use packages and namespaces to organize clusters of classes providing weak control for accessing members. Other languages that support generic programming and objects do not have a simple object model to support object-oriented features.

In this thesis the language MOOL is presented. MOOL is a class-based object-oriented language that supports modular programming and genericity.

The main goal in the design of MOOL was simplicity rather than efficiency. MOOL contains separated mechanisms for different concepts like classes and modules, which are unified in other languages. MOOL is not a pure object-oriented language where everything is an object. Non-object features like functions and modules are part of the language to enhance expressivity, to structure programs and to support code reuse.

Table of Contents

Keywords

Class

Inheritance

Interface

Generics

Modules

Objects

Object-oriented programming

Parameterized types

Polymorphism

Programming languages: Ada, BeCecil, BETA, C#, C++, Cecil, CLOS, Dylan, Eiffel, GJ, Jam, Java, Kevo, Mesa, MOBY, Modula-3, OCaml, Pizza, PolyTOIL, Self, Simula, Smalltalk, and SML.

Subtype

List of Figures

List of Tables

Acknowledgement

I would like to express my gratitude to my advisor Dr. Ryan Stansifer. His patience, support, and guidance during these past years helped me to accomplish this goal.

I would also like to thank the other members of my committee Dr. Pat Bond, Dr. Phil Bernhard, and Dr. Dennis E. Jackson for their valuable comments.

I'm grateful for the financial support I received from the Mexican Government and Instituto Tecnológico de Culiacán. Without their support I could not have completed this work.

Special thanks go to my family in México: To my mother, Lucia, who always encouraged me to pursue a professional career. To my father, Eulogio, who supported and helped me in many different ways. To my sisters Arminda, for being my role model, Maguie, Lety, and Norma for their help, my brothers Adolfo, Hernán, César, Javier, Eduardo, and Jesús for their unconditional love. To my niece Lucia Margarita and her dad Jorge for being there when I needed them.

I want to thank my husband, Ramon for helping me to accomplish this project and my daughters Naomi, Ana, and Lucia for their love and company.

Last but not least, I want to thank God from the bottom of my heart.

Dedication

This dissertation is dedicated to my little daughters

Naomi, Ana, and Lucia.

Chapter 1.

Introduction.

In the past few years the focus of many researchers has been the inclusion of polymorphism in the two popular object-oriented languages, Java and C# [BCK+ 01, BCK+ 03, BOSW 98a, CS 98, EKMS 97, MBL 97, OW 97, T 97, V 01b, and KS 01].

At the same time, the concept of module doesn't seem to exist in these two languages, in which the class is the only structuring mechanism for programs [GJSB 00, C# 01]. Modules are constructs used to build large programs, supporting encapsulation and information hiding and are of a different nature from classes. Some researchers have recognized the importance of having a module system in object-oriented languages [S 92, FF 98, BPV 98, AZ 01].

The lack of a module system in object-oriented languages leads to overloading the class with different purposes. And, the absence of parametric polymorphism obligates the programmer to implement non-natural solutions resulting in many problems.

These two problems motivated this work. The main goal of this dissertation is the design of a programming language that supports parametric polymorphism to

develop generic code, provides a module construct to safely create large programs, and supports object-oriented programming. Our main goal in the design is simplicity, i.e., keep the language as simple as possible.

The design of a programming language involves many decisions when selecting the features of the language and their interaction with each other. Many concepts are not included in the language. Out of the scope of this thesis are:

- Exception handling.
- Concurrency.
- Parallel programming.
- Implementation details.
- Formal semantic definition.

We have designed a *Modular Object-Oriented Language* called *MOOL*.

The language is designed to support the development of small or large programs. Small programs, which express the details of algorithms and data structures, can be created in a single module implementation with a standard predefined module interface. The language allows the definition of several modules and module interfaces to support “programming in the large”. Interconnected modules and module interfaces express the way the system is organized. Object-oriented programming is achieved by the use of classes and other features that are

part of the language. Parameterized types and type variables are also part of the language to support the definition of generic code.

1.1. Problem description.

In the 1970's modules were recognized as an important mechanism for structuring large programs. Programming languages which incorporate modules were designed to satisfy three important principles in the development of software: encapsulation, information hiding, and separate compilation. A module is a static entity that contains an interface to describe how a module can be interconnected with other modules. Modules are neither types nor extensible structures.

Object-oriented languages appeared with Simula and were popularized by Smalltalk and C++ in the 1980's. Object-oriented languages and methodologies have been widely used in the development of software over the past decade. Designers of object-oriented languages have decided to adopt only one structuring mechanism (the class) trying to reduce the number of concepts in the language. As a result, the class mechanism is overloaded with several functionalities and the concept of class is blurred. The absence of a module system in these languages does not allow one to express some concepts naturally.

Parametric polymorphism has proven to be a valuable feature that is not part of several widely used object-oriented programming languages. Several approaches

can be taken to implement generic code, but the solutions suffer from various problems.

Our challenge is to design an object-oriented programming language that supports the definition of modules to structure large programs and allows generic programming to write polymorphic code that can be used with several types.

1.2. Road map.

Chapter 2 presents concepts related to modularity. It describes the module system of some programming languages as well as other mechanisms used in some object-oriented languages that do not have a module system.

Chapter 3 includes a classification of object-oriented programming languages and describes the most important concepts related to object-oriented programming languages. The chapter contains examples of these concepts in several languages.

Chapter 4 explores concepts related to generics. It describes different kinds of generic code and different translation approaches implemented in different languages. It also presents examples of generic programming in several programming languages.

Chapter 5 sets up the goals of the new language design. It analyzes the concepts that need to be included in the language.

Chapter 6 introduces MOOL -Modular Object-Oriented Language. In this chapter, the general features of MOOL are described as well as the elements that are part of the language. The formal description of the grammar of MOOL is presented in an appendix.

Chapter 7 is dedicated to the assessment of MOOL. It revisits the issues mentioned in previous chapters as problems in some other languages and approaches them in MOOL.

Finally, chapter 8 contains our conclusions, contributions, limitations of MOOL, and future work.

Chapter 2.

Modules.

Modules are the abstractions used to structure large programs. Modules emerged in several programming languages after Parnas' seminal papers [P 72, P 72b]. Modules were introduced in the 70's in Mesa [MMS 79] and popularized in the 80's by Modula-2 [W 83]. Many programming languages that were designed later also incorporated this concept although they used different names for it. Packages, clusters, and structures are the names of modules in Ada [ADA 80], CLU [L+ 81], and Standard ML [MTH 90]. Recently, designers of modern object-oriented languages have tried to unify this concept providing only the class to structure programs.

This chapter contains the definition of concepts related to modules in programming languages. It also contains a description of different kinds of modules and some examples of the module system of several programming languages.

2.1. Concepts.

Modules in programming languages are recognized as an important mechanism for structuring large programs. They allow decomposing a system into smaller units that are easier to understand and manipulate. Modules encapsulate abstractions and provide a mechanism for protection. Modules are conceptually related to another concept: separate compilation. They are self-contained units and can be used as compilation units [Ca 89].

In this section we provide definitions for concepts related to modularity.

Modularization is a process in which a program is partitioned into a group of independent modules that expose their functionality and hide their internal structure. "*Modularization* is the process of decomposing a program in small units (*modules*) that can be understood in isolation by the programmers, and making relations between these units explicit to the programmer." [L 94]

Modular programming is a programming discipline that follows Ingalls' modularity principle: *"no part of a complex system should depend on the internal details of any other part."* [I 78]

A **compilation unit** is a unit that can be received by the compiler to translate it into target code.

"***Separate compilation*** is the process of decomposing a program in small units (*compilation units*) that can be typechecked and compiled separately by the compiler." [L 94]

A **module** is a static unit used to encapsulate elements, hide information and separate compilation. Modules have two parts: a *module interface* and a *module implementation*. Modules allow us to divide programs in smaller units. We can develop, check, deliver, optimize, and maintain these units separately. Several modules of a system can be developed in parallel if their module interfaces are provided.

A **module interface** is a specification of the elements that are provided by the module. It contains a subset of the definitions of the module implementation. A module interface describes only those elements that are not hidden. A module interface describes how the module can be plugged into another module to interact with it.

A **module implementation** contains at least the definition of the elements listed in its interface. Elements not listed in the interfaces are hidden from users of the module. Separating the module interface form the implementation makes possible to hide some information, which is necessary to avoid code dependencies.

Three concepts are closely related to modularity. They are: abstraction, encapsulation, and information hiding. *Abstraction* is the ability to represent only the important aspects of an entity but not its details. *Encapsulation* is a concept that relates aggregation and information hiding. Aggregation allows the definition of a set of elements in a unit. *Information hiding* is a design principle proposed by Parnas [P 72]. It allows making visible only some of the elements defined in a unit.

Encapsulation facilitates, but does not enforce, information hiding. Information hiding restricts the elements that can be seen in a unit helping to prevent code dependencies. In figure 2.1 we show how these three concepts are related in the definition of a module interface and implementation.

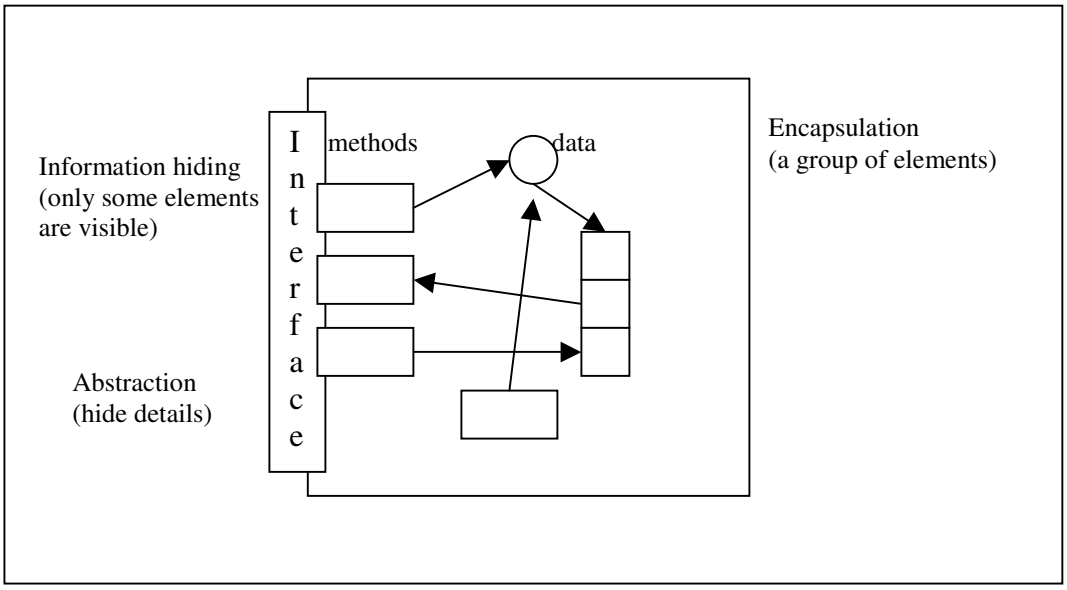

Figure 2.1. Abstraction, encapsulation and information hiding.

2.2. Modularity goals.

After a "software crisis" was recognized in the 1970's, many researchers started searching for solutions. Parnas proposed a technique for software module specification in which the goals of the specification scheme were based on the information hiding principle [P 72]:

1. The specification must provide to the intended user *all* the information that he will need to use the program correctly, *and nothing more*.
2. The specification must provide to the implementer, all the information about the intended use that he needs to complete the program, and *no additional information*; in particular no information abut the structure of the calling program should be conveyed.

It is not easy to split large programs into modules. Module boundaries depend on what we want to achieve, maintainability, performance, etc. The criteria used to modularize a system affect the time to develop the system as well as its flexibility and comprehensibility [P 72].

Two concepts related to the decomposition of a system into modules are *coupling* and *cohesion*. Modules should be as independent as possible from other modules, i.e., coupling should be minimized. Modules should enclose closely related data types, i.e., cohesion should be maximized.

The goals in modularizing a system can be described as follows:

- Maximize encapsulation. Data and procedures that manipulate this data must be in the same unit.
- Minimize information leaks in units. Procedures that access instance variables must be in the same implementation unit.
- Maximize information hiding. Restrict visibility and access to data and data types defined in units. Define separated units to describe and implement code. Implementation units can implement one or more interfaces. Interface units describe a set of types and the public operations allowed in those types. Two kinds of interfaces can be

generated: *client interfaces* describe the information needed by clients to use the unit, and *specializer interfaces* describe the information provided for designers to specialize code.

- Provide default values to avoid run-time errors due to initialization.
- Restrict and control access and visibility of members.
- Separate compilation. Compilation units can be compiled at different times, but their compilations are not independent of each other if either unit accesses or uses any entities of the other.

2.3. Kinds of modules.

Modules are the units of decomposition in large systems but modules serve some other purposes. Cardelli [Ca 89] describes three kinds of modules: any part of a program that could be reused, a collection of routines that maintains an invariant, and collections of related data types with their operations.

In this section we describe several kinds of modules that are used to solve specific problems. The kinds of modules are:

- Modules and interfaces.
- Libraries.
- Shared data area.
- Generic modules.
- Parametric generic modules.

2.3.1. Modules and interfaces.

A system can be described as a set of interconnected modules where the functionality of each module provides part of the functionality of the whole system. A module can have several interfaces that define the connections with other modules. Figure 2.2 shows a graphical representation of a set of interconnected modules that describe the architecture of a system for a convenience store.

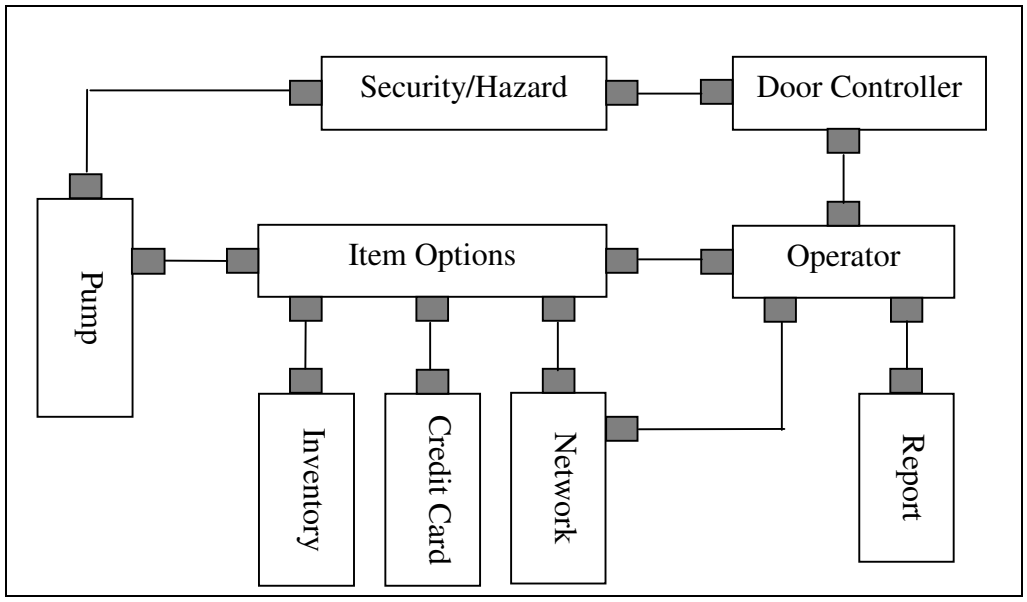

Figure 2.2. A software architecture design.

In this example, the system was modularized into 9 modules. Each module contains one or more interfaces represented by small gray squares, which are used to describe the interconnection with other modules.

2.3.2. Libraries.

A library is a kind of module that encapsulates a set of constants and functions. Libraries do not contain any data structure and they are, usually in compiled form, for linking with other programs. The most common example of a library is the one that contains a set of mathematical functions. Figure 2.3 shows an example of a library definition.

```
INTERFACE Math;
  CONST PI = 3.14159;
        E = 2.71828;
  PROCEDURE Sin (x : REAL) : REAL;
  …
END Math.
```

```
MODULE Math;
  PROCEDURE Sin ( x: REAL) : REAL =
  BEGIN
     (*  Sin implementation *)
  END Sin;
  …
END Math.
```

Figure 2.3. A Math library interface and implementation in Modula-3.

2.3.3. Shared data areas.

Modules that contain data structures but no functions are in this category. This kind of module allows the definition of some data structures that are going to be shared among several subprograms. The data structures defined in the package are available for those subprograms that use the package. Figure 2.4 shows an example of a module that defines a data structure that can be used by any other module that imports it.

```
package SHARED_BUFFER is
   MAX : INTEGER := 50;
   IN_PTR, OUT_PTR : INTEGER range 0..MAX := 0;
   BUFFER : array (0..MAX) of CHARACTER := (0..MAX = > ' ');
end SHARED_BUFFER;
```

Figure 2.4. A module with a shared data area in Ada.

2.3.4. Generic modules.

The Ada package shown in figure 2.5 encapsulates a data structure (STACK) and provides an interface to access it. A program can use the package by including a *use* declaration. However, the package describes a single data structure and it is not possible to have more than one stack in a program.

```
package STACK is
   procedure PUSH (ELEM : in INTEGER);
   procedure POP (ELEM : out INTEGER);
   function EMPTY return BOOLEAN;
end STACK;
```

```
package body STACK is
   STORE : array(1..100) of INTEGER;
   TOP : INTEGER 1..101 :=1;
   procedure PUSH(ELEM : in INTEGER) is
   -- PUSH implementation
   end PUSH;
   …
end STACK;
```

Figure 2.5. A package as a data structure manager.

When a program needs more than one data structure, it is not a good decision to copy the package to create a new one. Ada provides a way to define a template for packages that can be instantiated as needed. Figure 2.6 shows a

generic package in Ada. The package specification, shown in the left part of figure 2.6, starts with the word *generic,* that means the package defines a template. The package body is exactly the same as the one defined in the right part of figure 2.5.

```
generic
package STACK is
   procedure PUSH (ELEM : in INTEGER);
   procedure POP (ELEM : out INTEGER);
   function EMPTY return BOOLEAN;
end STACK;
```

```
package SCORE_STACK is new STACK;
package OPER_STACK is new STACK;
```

Figure 2.6. A generic package in Ada.

Two instances of the generic package are defined in the right part of figure 2.6. Every instantiation of the generic package creates a copy of the data defined in *STACK* and the procedures can be shared by all instances.

In this example, the elements of the stack are restricted to type INTEGER; this generic does not have a way to generalize the type of elements that the *STACK* can handle. The next section shows how this is accomplished.

2.3.5. Parametric generic modules.

Ada's generic packages can have several types of parameters. A type-independent package can be defined specifying a generic type parameter. This type parameter can be used in the package. An example of a generic package that is type independent is shown in figure 2.7. The left part shows a generic stack

specification where two parameters are defined: *MAX* is the maximum number of elements the stack can contain and *ELEMENT* is the type parameter that will receive the specific type when an instance of the package is created. It is possible to restrict the types that can be used as arguments to instantiate the generic package. In this example only two operations (assignment and equality comparison) are available for *ELEMENT* within the package.

```
generic
   MAX : INTEGER := 100;
   type ELEMENT is private;
package STACK is
   procedure PUSH (ELEM : in ELEMENT);
   procedure POP (ELEM : out ELEMENT);
   function EMPTY return BOOLEAN;
end STACK;
```

```
package body STACK is
   STORE : array(1..MAX) of ELEMENT;
   TOP : INTEGER 1.. MAX+1 :=1;
   procedure PUSH(ELEM : in ELEMENT) is
   -- PUSH implementation
   end PUSH;

   …
end STACK;
-------------------------------------------------------------------------
package body MAIN is
    package SCORES is new STACK(100, INTEGER);
    package INITIALS is new STACK(32, CHARACTER);
    …

end MAIN;
```

Figure 2.7. A parametric generic package in Ada.

Two instances of the generic package are created in the package *MAIN* shown in right bottom part in figure 2.7. The package named *SCORES* is an instance of the generic package *STACK* where the maximum number of elements is 100 and the type of elements it stores are INTEGER. *INITIALS* is also an instance of *STACK* with a maximum capacity of 32 elements of type CHARACTER.

2.4. Examples in some programming languages.

The module system of a programming language facilitates the design and reuse of software. A module is a static entity that, once defined, cannot be changed. A programming language *supports* modular programming if it provides facilities to create and express a modular structure. If the language does not provide these facilities, it is still possible to develop a modular structure but greater effort will be required. We can say that almost any language *permits* modular programming but only those with a module system *support* it. Many programming languages like Mesa, Modula-2, Ada, Modula-3, and Oberon have a module construct. Leroy [L 00] suggested that the design of a module system is independent of the base language used and a general module system can be applied to a variety of languages. Although he recognized that the code structuring features of object-oriented languages overlap with a module system.

The major features of a module system are:

1) Encapsulation
2) Information hiding
3) Separate compilation.

In this section we show the features of the module system of several languages.

2.4.1. Ada packages.

Ada modules are called packages. A package is a program unit that contains a group of entities. It has two parts: a package specification and a package body, which is the implementation. Ada packages support data abstractions when they contain a declaration of a type and a set of operations (subprograms) on that type. The package specification exposes the information available for clients. The package body provides the implementation of the data abstraction. Ada packages are considered second-class objects because they are not types. Some examples of Ada packages are shown in sections 2.3.3, 2.3.4, and 2.3.5.

2.4.2. Modula-3 modules.

Modula-3 [N 91] provides two basic program units: modules and interfaces, which can be generic. A collection of both modules and interfaces defines a program. A module is a unit that implements one or more interfaces that are exported by the module. A module defines a block where all declared entities are visible inside itself as well as all entities declared in imported interfaces.

An interface is a unit that contains a group of declarations where variable initialization is constant and procedure declarations specifies only its signature. They are used to hide information and restrict access to members of the module. Interfaces are separated from their implementation. Only the elements listed in the

interface are available for clients. Since interfaces do not contain implementation details, code dependencies are avoided.

A module imports an interface to make its entities available. A module exports an interface if it provides bodies for its procedures. A module restricts the visibility and accessibility of its members using interfaces to export them. When a module does not specify an exported interface, all the elements of the module are exported. Figure 2.8 shows an example of an interface and a module implementation for a linked list.

```
INTERFACE LinkList ;
  TYPE
    LinkedList  = OBJECT
       METHODS
          Add( o: ROOT);
          Empty( ): BOOLEAN;
   END;
END LinkList.
```

```
MODULE LinkList;
 TYPE
     Linkable = OBJECT
       node : ROOT;
       next : Linkable := NIL;
    END;
    LinkedList =  BRANDED OBJECT
      head : Linkable := NIL;
      METHODS
      Add(o: ROOT) := AddProc;
      Empty( ) := EmptyProc;
   END;
   PROCEDURE AddProc( self: LinkedList; o: ROOT) =
   VAR  x : Linkable;
   BEGIN
        x := NEW (Linkable, node:=o; );
        x.next := head;
        self.head := x;
   END AddProc;
   PROCEDURE EmptyProc( self: LinkedList ; ) : BOOLEAN =
   BEGIN
       RETURN self.head = NIL;
   END EmptyProc;
BEGIN
END LinkList.
```

Figure 2.8. An interface and module implementation in Modula-3.

2.4.3. SML structures, signatures, and functors.

SML is a functional programming language that supports modular programming [MTH 90]. The module system of SML is considered one of the most powerful due to its treatment of parameterized modules as functors [L 00].

SML module system has structures, signatures, and functors. A *structure* is a unit that encapsulates a collection of types and values. A *signature* specifies the type of a structure, i.e., it is a "structure type" and can restrict the accessibility of the members of a structure. The types and values declared in a structure can be referred using qualified names, i.e., using the dot notation *structureName.identifier*. A *functor* is a function that receives a structure as argument and produces a structure as result, i.e., a function from modules to modules. Figure 2.9 shows an example of a signature, a structure, and a functor in SML.

The left part of figure 2.9 contains the definition of two signatures *MONOID* and *POWMON*, and a functor called *Pow*. Functor *Pow* receives as a parameter a structure of type *MONOID* and gives as a result a structure of type *POWMON*. In the right part, we see the definition of two structures *RealAdd* and *IntMul* which type is *MONOID*. Two new structures, *S* and *I,* are obtained by applying the functor *Pow* with arguments *RealAdd* and *IntMul*. These two structures *S* and *I* have type *POWMON*.

```
signature MONOID =   sig
    type set ;
    val ope : set * set -> set ;
    val one : set ;
  end;

signature POWMON =   sig
    include MONOID ;
    val pow : set * int -> set ;
  end;

functor Pow (Monoid : MONOID) : POWMON =
  struct
    type set = Monoid.set ;
    val ope = Monoid.ope ;
    val one = Monoid.one ;
    fun pow (a,n) = if n=0 then one
                    else ope (pow(a, n-1),a) ;
  end;
```

```
structure RealAdd =    struct
    type set = real;
    fun ope (r1 : real, r2 : real) = r1 + r2 ;
    val one = 0.0;
  end;

structure IntMult =   struct
    type set = int ;
    fun ope (i1 : int, i2 : int) = i1 + i2 ;
    val one = 1;
  end;

structure S = Pow (RealAdd) ;
S.pow(4.5, 0);
S.pow(37.68, 2);

structure I = Pow (IntMult) ;
I.pow (4, 3) ;
I.pow(4, 0) ;
```

Figure 2.9. Signatures, structures, and functors in SML.

SML modules are first-class values. They can be passed as parameters and returned as results in functors and they can be stored in data structures.

A problem of the SML module system is related to separate compilation. As stated in its definition, "*ML is an interactive language*" [MTH 90, page 1]. Leroy attempts to apply the separate compilation technique found in Modula-2 to the SML module system [L 94]. This cannot be directly applied because the use of transparent types specifications prevents the complete type specification of signatures, which is required to detect type clashes in modules. Leroy proposed the use of *manifest type declarations* to provide enough type information for separate compilation.

2.4.4. Objective Caml.

Objective Caml (OCaml) is a general-purpose language that supports functional, imperative, and object-oriented programming styles [R 02]. OCaml supports two models to structure programs: the parameterized module model and the object model.

The parameterized module model allows decomposing a program into software units, which are called modules. They can be developed independently and compiled separately. Modules can be also parameterized increasing the possibility of code reuse.

There are two ways to create modules: as *compilation units* and using the *module language*. A compilation unit is created with two files with different extensions: an interface file (.mli) and an implementation file (.ml). Modules as compilation units have some drawbacks: a one-to-one relation between modules and files exist making impossible to use several implementations of an interface, and nested modules are not supported. The module language of OCaml is similar to the module system of SML. It contains two kinds of modules: signatures and structures to define interfaces and implementation respectively. A structure can be constrained by a signature making accessible to clients of the module, only those elements listed in the signature. Functors are also part of the module language of OCaml and they have the same functionality than in SML. Figure 2.10 shows an

interface with its implementation. The example is based on an example from [CMP 00, page 408].

```
module type STACK =   sig
     type 'a t
     exception Empty
     val create : unit -> 'a t
     val push: 'a -> 'a t -> unit
     val pop: 'a t -> 'a
  end  ;;

module AStack =  struct
     type 'a t = {mutable top : int; mutable store : 'a array }
     exception Empty
     let increases s x = s.store <- Array.append s.store ( Array.create 5 x)
     let create () =  { top=0 ; store = [ | | ]  }
     let push x s =
          if s.top >= Array.length s.store then increase s x;
          s.store.(s.top) <- x;
          s.top <- succ s.top
     let pop s
          if s.top = 0 then raise Empty
          else (s.top <- pred s.top ; s.store.(s.top))
  end ;;

// instance of AStack constrained by STACK signature
module Stack1 = ( AStack : STACK ) ;;

let s1 = Stack1.create();;
Stack1.push 2 s1 ;;
```

Figure 2.10. Modules in OCaml.

2.4.5. Java packages.

The Java programming language does not have a traditional module system. Classes and packages in Java are used to support language features that are part of the module system in other languages. Java provides classes to structure programs

and packages to group related classes and interfaces under a common name. Java packages serve three main purposes:

- Define package scope. Classes defined in the same package may share some of their information. Packages can be nested to organize related packages, but no special access is provided for them. Package scope applies only to the package itself and no other nested packages.
- Define namespaces. A package can contain several definitions of interfaces and classes. Packages have a one-to-one relation with file directories. They are used to create naming contexts.
- Import. The elements of a package can be accessed using fully qualified names or they can be imported. A program can import all or part of a package. The use of a package prefix ensures that names in one context do not conflict with names in another context.

The Java platform has several standard packages that define the core Java classes. Figure 2.11 shows an example of a group of classes that belong to a package.

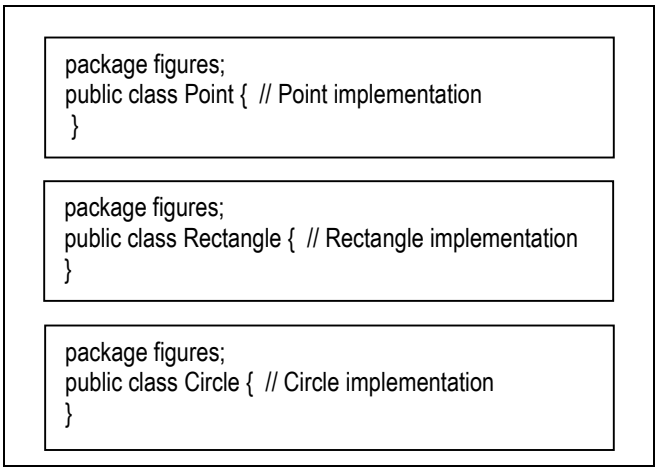

Figure 2.11. Java packages.

Three classes are defined in the package, which is named *figures*. In each file, the first statement declares that all classes and interfaces in the file are part of the package. When many classes are defined in file, only the one whose name coincides with the file name can be annotated as **public**. The name of the package is prefixed to each element contained in the package. Classes within the package can refer to each other members. Fields and methods that are not annotated as **private** can be used in all the code within the package. Class members' access is by default **package**, which means they can be used by other classes in the package. Some problems of the Java package mechanism are:

- Name collision. The same name can be given to different packages.
- Packages cannot completely control the access to their member classes.
- Packages do not have interfaces and cannot provide different views.

2.4.6. C# namespaces and assemblies.

C# provides files, namespaces, and assemblies to organize the source code. A *namespace* is a logical structuring mechanism that can contain several classes and interfaces. They are used to avoid the use of long class names. A namespace can be imported making accessible all the classes and interfaces that it contains. An *assembly* is an executable file (.exe or .dll) generated by the compiler. An assembly is used to pack and deploy a component. Figure 2.12 shows an example borrowed from [C# 01 p. 45].

```
// class library with a single class
// HelloLibrary.cs
namespace Csharp. Introduction {
  public class HelloMessage {
    public string Message {
      get { return “hello, world”;  }
    }
  }
}
```

```
// application
// HelloApp.cs
using Csharp. Introduction;
class HelloApp
{
  static void Main( ) {
    HelloMessage m = new HelloMessage ( );
     System.Console.WriteLine (m.Message);
  }
}
```

Figure 2.12. C# namespaces and assemblies.

The left part of figure 2.12 shows an example of a namespace named *Csharp.Introduction*, which contains only a class named *HelloMessage*. The fully name of this class is *Csharp.Introduction.HelloMessage.* The right part of figure 2.12 shows an application that uses the class *HelloMessage* which is available without its fully qualified name because a using namespace directive imports it.

These two files can be compiled to generate a class library and an application that uses that library.

2.4.7. Dylan.

In Dylan a module defines a namespace. It contains typically several functions and classes. A module definition can have three kinds of clauses: export, create and use. An *export clause* specifies which names are exported. Exported elements like classes, slots, variables, and functions are available for users of the module. A *uses clause* describes the modules used by the module being defined. A *create clause* specifies the names declared and exported by the module. An example of two module definitions borrowed from [S 97] is shown in figure 2.13.

```
define module graphics
  use dylan;
  create draw-line,
         erase-line,
         invert-line,
         skew-line,
         frame-rect,
         fill-rect,
         erase-rect,
         invert-rect;
end module graphics;

define module lines
  use dylan;
  use graphics,
    import: {draw-line,
             erase-line,
             invert-line,
             skew-line};
end module lines;
```

Figure 2.13. A module definition in Dylan.

The module *graphics* uses the module called *dylan*, which contains all the basic language primitives. The module *lines* uses modules *dylan* and *graphics* but the import declaration specifies that only some elements of graphics are available (*draw-line, erase-line, invert-line,* and *skew-line*).

Chapter 3.

Object-Oriented Concepts.

Objects, classes, and inheritance are some of the concepts related to object-oriented programming. They emerged in the 1970's with the programming language Simula and Smalltalk [DN 81, I 78]. Now, the object-oriented concept refers to both a methodology for software design as well as a programming language feature. Although there is no consensus about the exact meaning of that concept, some features are commonly recognized to be part of it. Most object-oriented programming languages provide mechanisms for: encapsulation, inheritance, dynamic dispatch, open recursion, and inclusion (subtype) polymorphism. These features have a lot of variations and their combination gives a unique flavor to each language.

In this chapter we present a classification of object-oriented programming languages and review the different variations of the most important concepts that distinguish an object-oriented language. We also show examples of how these concepts are used in some specific programming languages.

3.1. Classification of object-oriented languages.

Three varieties of object-oriented languages are distinguished in figure 3.1, they are: passive, active, and multimethod languages. Bruce [B 02] uses a similar categorization with another naming convention: class-based, object-based and multimethods languages. We adopted different names to prevent confusion with the categories proposed by Wegner [W 87] that uses the terms object-based language and class-based language for non-object-oriented languages.

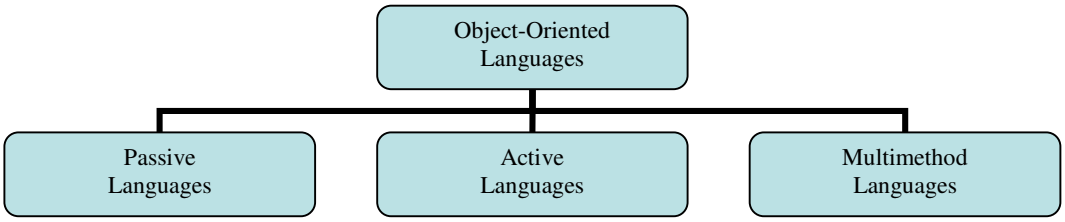

Figure 3.1. A classification of object-oriented languages.

A brief description of each kind of object-oriented language is presented in this section.

3.1.1. Passive languages.

Passive languages are also known as class-based languages because they contain a class construct to describe the implementation of a group of objects. Objects are instances of the class. All the objects generated from a class share the

same behavior, which cannot be changed at runtime. The class defines a set of values, called instance variables or fields, and a set of operations over those values, called methods. Classes can contain also special functions called constructors to generate and allocate instances of that class. Subclasses are specializations of classes and they are defined using inheritance. In many languages classes and subclasses generate types and subtypes.

Some examples of class-based languages are Java [AG 98], C# [C# 01], C++ [S 91], Eiffel [M 92], OCaml [R 02], MOBY [FR 99b].

3.1.2. Active languages.

Active languages are also called classless or object-based languages because they do not contain a class construct to define objects. Objects are formed directly by constructing concrete objects called prototypes or exemplars. In these languages, every object has its own behavior, which is set when the object is created and, like fields, the behavior can change at runtime. Since no class mechanism exists, these languages provide different mechanisms to derive new objects from existing ones.

Prototype-based languages are a kind of active language. These languages allow the user to generate “prototypical” objects that are used to create new objects. They also allow dynamic update of methods so that an object can change its behavior at runtime. Cloning is an operation to create a new object from a

prototype. All clones have the same structure but new features can be defined using extension or delegation. Extension works like inheritance in classes. An object created by extension inherits everything and it is independent of its parent. Delegation allows an object to delegate certain operations to another object. An object created by delegation retains ties to its prototype parent such that changes to the prototype will be visible to the delegate and vice versa.

Self [US 87] and Kevo [T 93] are prototype-based object-oriented languages with an associated programming environment. Objects in Self are made up of slots, which can represent state or behavior. The environment provides a graphical representation for objects. In this graphical representation, every slot has an icon that represents data (constant or assignable) or behavior. The state of an object can be altered only by passing messages to it. The types of variables are not restricted so static typechecking is not possible. An example of an object in Self, borrowed from [WS 03] is presented in the left part of figure 3.2. Another example of a prototypical object called *point* written in Kevo, is presented in the right of figure 3.2.

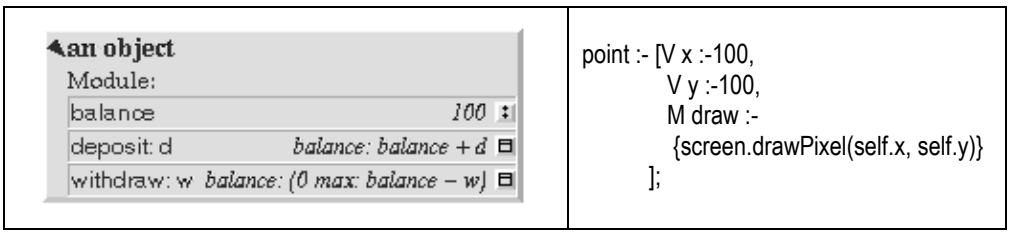

Figure 3.2. Prototypical objects in Self and Kevo.

3.1.3. Multimethod languages.

Some object-oriented languages are not distinguished by the mechanism they use to create objects, but by how they deal with dynamic dispatch. Dynamic dispatch is divided into single and multiple dispatch.

In languages with multiple dispatch the object itself does not contain the methods. The methods are implemented as functions, which can have the same name as other functions but different parameters. A group of functions with the same name and different parameters are called overloaded generic functions. A theoretical study of this model of message sending appears in [CGL 95].

Multimethod languages are object-oriented languages with multiple dispatch. They use overloaded generic functions to support dynamic method invocation. When a message is sent, the name of the message and the type of the arguments are used to select which overloaded generic function is going to be executed. If there is only one match, it is selected but if there were more than one, the selection of the method would be the one with the best fit. Some ambiguities can arise in the selection of the method. Sometimes it is the programmer responsibility to solve any ambiguity and sometimes the language contains a mechanism to solve the ambiguity. A similar problem emerges in programming languages with multiple implementation inheritance and similar solutions are provided.

Multimethod languages do not encapsulate data and functions together, they are separated, and sometimes they are encapsulated in modules. This separation of data and functions increases the expressiveness of the language but it breaks encapsulation. Most multimethods languages are classless but some of them have a class construct that defines only data members, e.g., Dylan [S 97]. Proponents of multimethods languages argue that languages with multiple dispatch are more expressive than single dispatched languages because multiple dispatch is more symmetric than single dispatch. Multiple dispatch solves the problem with binary methods allowing covariant redefinition of parameters, and procedures. Single dispatched methods and overloaded functions can be generalized by multiple dispatch [CL 97].

Some examples of multimethod languages are BeCecil [CL 97], Cecil [C 98], CLOS [BDG+ 88], and Dylan [S 97].

Figure 3.3 shows an example of an object definition and an overloaded generic function written in BeCecil [CL 97].

BeCecil is a statically typed, classless object-oriented language with multiple dispatch and multiple inheritance where classes and types are separated in two different hierarchies. It supports a prototype-based model unifying classes and objects. BeCecil is a core language designed as a subset of Cecil but it does not include all its features. Cecil is a multimethod language with parameterized types and a module system [C 98].

```
object Point_rep
Point_rep inherits any
object ColorPoint_rep
ColorPoint_rep inherits Point_rep
ColorPoint_rep inherits Color_rep              -- multiple inheritance

object x
x inherits GenericFun_rep
x has storage(p@Point_rep) := 0       -- instance variable
x has method (p@Point_rep) {
  x_rep (p)                           --  returns field  x, which has type  Int_rep
}
-- same code of object x to define instance variable y

object equal
equal inherits GenericFun_rep
equal has method (v1@Int_rep, v2@Int_rep) {
  v1 = v2                          -- compare v1 with v2 returns boolean
}
equal has method (p1@Point_rep, p2@Point_rep) {
  and (equal(x(p1), x(p2)), equal(y(p1), y(p2)))
}
```

Figure 3.3. Overloaded generic functions in BeCecil.

In the example of figure 3.3, an object called *Point_rep* is defined. It is derived from *any*, which is a special object from which almost all objects inherit. Another object called *ColorPoint_rep* is defined with multiple inheritance. A generic function is a collection of multimethods. Generic functions *x* and *equal* are objects that inherit from *GenericFun_rep*, which is a predefined object derived from *any*. A generic function is extended with a *has* declaration. The generic function *equal* is extended with two methods in the example.

3.2. Abstract data types and objects.

"A type is a (partial) description of behavior- a statically verifiable interface" [B 92]. An abstract data type (ADT) is a type that contains a set of values and a set of operations on those values. The structure or representation of the values and the implementation of the operations are not important to the client and can be hidden.

Transparent data types expose their representation to clients, while opaque data types hide their representation to clients. Hiding the representation from the client avoids code dependencies; thus the implementation can be changed without affecting the client code. Programming languages that support ADTs usually contain the following:

- A mechanism to separately define the declaration of an ADT from its implementation. These two parts can be placed in different units.
- Separate compilation for the program units that declare and implement the ADT.
- A mechanism to restrict visibility and access to the implementation part.
- A mechanism to allow the client to access the ADT.

On the other hand, an object contains state (a set of fields) and behavior (a set of operations, called methods). The operations of an object are executed by

passing a message to the object, which is the name of the method to be executed. A class is a template to define a group of objects. An object is an instance of a class.

Objects and abstract data types have some common properties but they also have some properties that make them different. For example, the use of inheritance in the definition of objects makes encapsulation more complex than simple data abstraction. Objects carry with themselves the set of operations they can execute and select the method to execute dynamically. On the other hand an abstract data type stores the operations in a module and they are statically located. An object can be replaced at runtime by another one that has the same interface.

3.3. Dynamic dispatch.

The selection of the method to be executed can be resolved at compile time or at runtime. Object-oriented languages use a mechanism known as dynamic dispatch for the selection at runtime; the method selected can change during program execution. Static binding takes place at compile time and remains unchanged in the execution of the program.

Dynamic dispatch is an important feature of object-oriented languages. The definition of subclasses allows redefining methods that were inherited from superclasses. Dynamic dispatch allows selecting dynamically the method to execute depending on the runtime type of the object that receives the message, in

languages with single dispatch, or on the runtime types of one or more arguments, in languages with multiple dispatch.

Dynamic dispatch is the default in Java while in C# and C++ must be explicitly declared by making the methods virtual.

3.3.1. Single dispatch.

In programming languages with single dispatch, when a method is called the selection of the method body is based on the runtime type of the designated object who receives the message. Figure 3.4 shows the definition of a class and a derived class which are going to be used in this example.

```
public class Point {
   int xval, yval;
   Point(int x, int y) { xval = x; yval = y;}
   public void move(int dx, int dy) { xval +=  dx; yval += dy;}
   public String toString () { return "x = "+xval+ ",  y = "+yval;}
}

public class ColorPoint extends Point {
   String cval;
   ColorPoint(int x, int y, String c) { super(x,y); cval = c);
   public void setColor (String c) { cval = c;}
   public String toString () { return super.toString + ",  color = "+cval;}
}
```

Figure 3.4. A class and a subclass definition in Java.

A common approach to select the method to execute is described as follows. When an object receives a message, it looks for the method in its method table, if

the method is not there, then it goes up into its parent method table. This process is repeated until the method is found. An example of a dynamic method invocation is presented in figure 3.5.

```
…
  Point point1 = new Point(1,1);
 System.out.println( point1.tostring()); // prints   x =1, y = 1

 point1= new ColorPoint(5,5,"red");
 System.out.println( point1.tostring());   // prints   x =5,  y = 5,  color = red
```

Figure 3.5. Dynamic method invocation with single dispatch.

In the previous example, when the variable *point1* contains a *Point* object, the method *toString* executed is the one defined in *Point* class. On the other hand when variable *point1* contains a *ColorPoint* object, the method *toString* executed is the one in *ColorPoint* class.

C++, Java, C#, and BETA are programming languages with single dispatch.

The BETA programming language has a different strategy for method lookup [MMN 93]. When a message is sent, the execution of the method starts at the top element of the hierarchy of objects. If the method is found it starts execution. If the method is not found then it will go into the object hierarchy looking for it. An example that illustrates this process is presented in figure 3.6

There are three patterns in figure 3.6, *Point*, *ColorPoint* and a pattern to use those two patterns. Patterns are delimited by (# #). Pattern *ColorPoint* is defined with inheritance using *Point* as parent, and it overrides the pattern *Init*. The

sentence *&colorpoint1.Init;* calls to execution *Init* of *ColorPoint*. This execution of *Init*, will execute first the actions defined in its superpattern *Point*, initializing *X* and *Y* with 0. The *inner* sentence following the initialization indicates that possible actions are going to be executed in a subpattern, so the control goes to the *Init* of *ColorPoint* where the initialization of *color* takes place. In this case, the *inner* part of *Init* in *ColorPoint* is an empty action because no subpatterns exist. This strategy to start looking for the method to be executed in the top of the hierarchy allows to preserve the behavior defined in parent classes, because overriding methods must execute first the methods they override.

```
Point: (#
   X,Y : @ integer;
   Init :< (# do 0->X; 0->Y; inner #);
#);
ColorPoint : Point (#
   color : @ integer;
   Init :< (# do 0->color; inner #);
#);
(#
   point1 : @ Point;
   colorpoint1 :@ ColorPoint;
do &point1.Init;                {initializes point1 }
    &colorpoint1.Init;          {initializes colorpoint1 }
    …
#)
```

Figure 3.6. Dynamic method lookup in BETA.

3.3.2. Multiple dispatch.

In languages with multiple dispatch the selection of the method to be executed is based on the runtime type of one or more parameters. Multiple dispatch

is implemented by the definition of overloaded generic functions. A generic function contains many implementations of a method name with different parameter types. They are a set of overloaded functions.

Figure 3.7 shows an example where several generic functions are called. This example uses the code presented in figure 3.3.

```
object point1
point1 inherits Point_rep
x(point1) := 1
y(point1) := 1

object point2
point2 inherits Point_rep
x(point2) := 2
y(point2) := 2

var o1: object := point1
var o2: object : = point2
equal(o1,o2)                    --    compare (point1, point2) result is false
o1 := x(point1)
o2 := y(point 1)
equal(o1,o2)                     -- compare fields x and y of point1    result is true
```

Figure 3.7. Executing generic functions in BeCecil.

Two objects *point1* and *point2* are defined. They are derived from the object *Point_rep*. Their instance variables *x* and *y* are initialized, which are themselves objects derived from *GenericFun_rep*, and contain storage and a method to recover their value. The generic function *equal*, is called to execution twice. The first time object *o1* and *o2* contain *point1* and *point2* respectively, so generic function *equal* is called with parameters of type *Point_rep*, while in the second call (last line of

code) object *o1* and *o2* contain objects of *Int_rep* which are the result of the execution of the generic functions *x* and *y* with the same parameter *point1*.

3.4. The special variables *this* and *super*.

Object-oriented languages define two special variables *this*, sometimes called *self*, and *super*. Variable *super* refers to the immediate parent class (object) while variable *this* refers to the object that originated the message. These two variables have an important role in finding the right method to be executed at runtime.

When a method is called with *super*, *this* is still bound to the original caller. The use of *this* in a method invocation represents a late-bound thus a method call from inside an ancestor class may not be the method of the ancestor class, but the method overridden by some descendent class.

3.5. Inheritance.

Inheritance is a language construct that allows the definition of new objects based on existing ones. It is an important feature that encourages code reuse. Given the definition of an object (class or prototype) a specialization of it can be generated by specifying the differences with respect to the given one. The new class is called a subclass, derived class, or child class, and the extended class is called the superclass, parent, or base class.

Inheritance can be classified in several ways. Single and multiple inheritance are related with the number of classes used as parents in the definition of new classes. Single and multiple inheritance are explained in sections 3.6.1 and 3.6.2. Another kind of inheritance, known as mixin inheritance is detailed in section 3.6.3. Interface inheritance refers to the inheritance of types rather than implementation and it is explained in section 3.6.4.

3.5.1. Single inheritance.

In languages with single inheritance, the definition of a new class called a subclass is derived only from one class. The class hierarchy created by single inheritance corresponds to a tree. Figure 3.8 shows an example of single inheritance.

```
class Parent {   }
class Child1 extends Parent {   }
class Child2 extends Parent {   }
```

Figure 3.8. Definition of classes using single inheritance.

In this example three classes are defined. Classes *Child1* and *Child2* are subclasses of class *Parent*. They inherit everything from *Parent*. Inherited members of the Parent class can be redefined in any subclass. The graphic representation of single inheritance is a tree, as shown in figure 3.9.

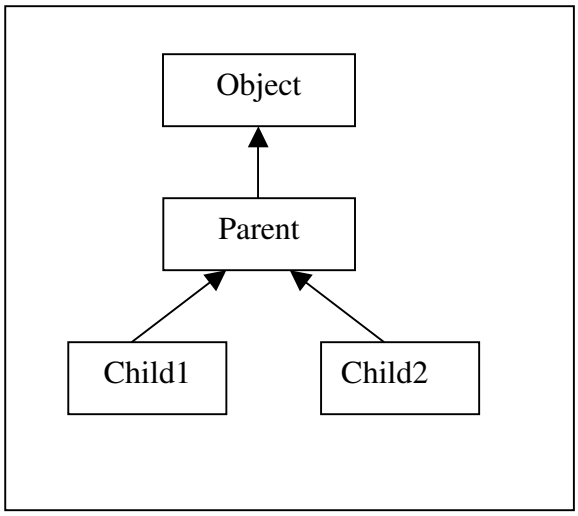

Figure 3.9. A hierarchy of classes with single inheritance.

The class *Parent* is implicitly derived from the class at the top of the hierarchy, which is a special class provided by the language. This special class is named *Object, ROOT,* and *any* in Java, Modula-3, and BeCecil respectively. Every class that doesn't have an explicit parent is implicitly derived from the class at the top of the hierarchy of classes.

Single inheritance has some advantages over multiple inheritance. The relationship between classes is simple and no ambiguities need to be solved. A disadvantage of single inheritance is that sometimes complex hierarchies of classes can not be easily expressed.

3.5.2. Multiple inheritance.

In multiple inheritance, a new definition is created by using two or more previous definitions. The graphical representation of the inheritance hierarchy is a Directed Acyclic Graph (DAG).

Complex hierarchies of classes can be expressed using multiple inheritance, and the main disadvantage is the problems that can arise with its use, e.g., the diamond problem. Two problems are well known, they are: name ambiguities for methods and redundant method calls. There is no satisfactory solution for these problems in current languages that support multiple inheritance because they violate some other principles. Figure 3.10 shows an example of a class defined with multiple inheritance in C++.

```
class link {  … }

class task : public link {
   virtual debug* get_d ();
}

class job : public link {
   virtual debug* get_d ();
}

class myClass : public task, public job{ ….  }

debug* d = s-> get_d();       // error! ambiguous method invocation

debug* e = s -> job::get_d();  // explicit delegation using class job

debug* f = s -> task::get_d();// explicit delegation using class task
```

Figure 3.10. Class definition with multiple inheritance.

In this example, class *myClass* inherits from two classes, *task* and *job.* These two classes have a common parent, class *link.* At the point in which function *get_d* is called a compile time error arises, because the call is ambiguous and the programmer must explicitly resolve this problem. In other languages like CLOS and Self, the mechanism for resolving such ambiguities is called “linearization”. A common approach to linearization is ordering left-to-right the inheritance graph and the arguments of the function to resolve ambiguities automatically. The disadvantage of this mechanism is that sometimes the selected method is not the one intended. Eiffel provides an explicit annotation named “feature renaming” that allows a class to rename one of the conflicting methods to disambiguate calls.

Figure 3.11 shows the class hierarchy created by the class definitions shown in figure 3.10.

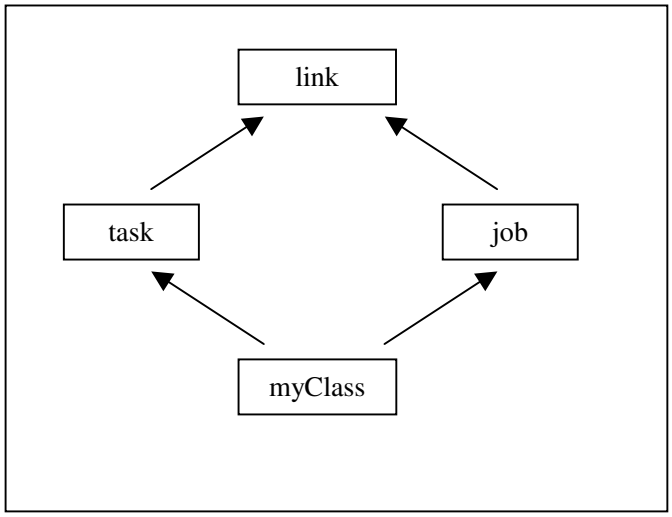

Figure 3.11. The diamond problem.

3.5.3. Mixin inheritance.

Single inheritance is not enough to express complex hierarchies of the real world and multiple inheritance adds complexity to the language which results in error-prone programs. A mechanism called *mixin* intends to solve these problems.

Mixins are not classes, but a mechanism to create new classes from existing ones. A mixin "class" is created with a specific purpose of being used to add properties to other classes. They are not meant to be instantiated so no constructors are provided. Mixins are not placed in the class hierarchy, so they stand alone, and can be reused in many different places [B 92].

Bracha and Cook [BC 90] proposed an inheritance mechanism based on composition of mixins that may be applied to superclasses to generate a new family of related classes in languages with single or multiple inheritance. The main advantage of this approach is that it does not depend on the linearization of ancestors avoiding the problems of languages with multiple inheritance. In languages with single inheritance, mixins can be used to model more complex hierarchies that are related with multiple inheritance.

The use of mixins in languages with single inheritance, allows having multiple inheritance without its problems. A graphical representation of mixin inheritance is shown in figure 3.12. In this figure a small circle represent a mixin and large circles represent classes. A mixin can be mix with a class that shares some properties creating a new class.

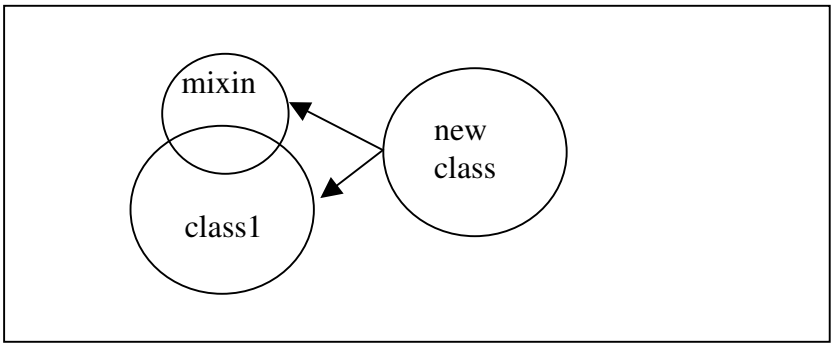

Figure 3.12. Mixin inheritance.

Jam [ALZ 03] is an extension of Java with mixins. Figure 3.13 shows an example of a mixin in Jam borrowed from [ALZ 03].

```
// mixin definition
mixin Undo {
  inherited String getText( );              //must be in the class to be mixed
  inherited void setText(String s);    //must be in the class to be mixed
  String lastText;
  void setText(String s) {
    lastText = getText( );
    super.setText(s);
  }
  void undo( ) {   setText(lastText);  }
}

class Textbox extends Component {   // a class with getText and setText
  String text;
  …
  String getText( ) { … }
  void setText(String s) { … }
}

class TextboxWithUndo = Undo extends Textbox { }  // new class
```

Figure 3.13. A mixin declaration and its use to produce a new class.

In this example, the mixin called *Undo* requires that the class to be mixed which contain the inherited elements. It defines a new field *lastText* and two methods *setText* and *undo*. Mixing the mixin *Undo* with the class *Textbox* creates a new class called *TextboxWithUndo*.

Mixins in Jam, define types, therefore classes created by mixin instantiation have both the type of the parent class and the type of the mixin.

3.5.4. Interface inheritance.

An interface specifies a set of abstract methods and properties. It does not contain default implementations. Interface inheritance is the process of defining a new interface with all the methods found in one or more old interfaces. Interfaces allow the definition of types that can have multiple implementations. Figure 3.14 shows an example of multiple inheritance of interfaces in Java.

```
public interface Student {
   Major getMayor( );
}

public interface Professor{
   Courses …
}

public class TeacherAssistance implements Student, Professor{

}
```

Figure 3.14. An example of multiple inheritance of interfaces.

Some languages with single implementation inheritance, like Java and C#, allow classes to implement multiple interfaces. In this case classes inherit only structure, not implementation. Multiple inheritance of interfaces avoid the problems of multiple inheritance of classes, but duplication of code is sometimes needed.

Some languages offer also abstract classes. An abstract class is a class that is not completely defined. It contains some abstract methods as well as some implemented methods. In Java abstract classes define types and are part of the class hierarchy. Interfaces and abstract classes in Java share some similarities but it is recommended to use interfaces to abstract classes [B 01]. A comparison of abstract classes and interfaces is listed next:

- Both interfaces and abstract classes define a type that permits multiple implementations. An interface is the best way to define a type to have multiple implementations.
- Abstract classes can be partially implemented; interfaces cannot.
- They are in different hierarchies.
- Abstract classes are easier to evolve than interfaces. (when an interface is updated it will break all existing classes implementing it.)
- Interfaces can act as mixins (mixin interface), abstract classes cannot.
- Existing classes can be easily retrofitted to implement a new interface. This is not the case for new abstract classes because they must be placed in the class hierarchy.

3.6. Polymorphism.

In the dictionary the definition of the word *polymorphism* is “the quality or state of being able to assume different forms.”

Two categories of polymorphism are recognized in programming languages: *universal polymorphism* where the same code works for many different types, and *ad hoc polymorphism* where different code is provided for every different type [CW 85]. Universal and ad hoc polymorphisms are subdivided to create four kinds of polymorphism. This is shown in figure 3.15

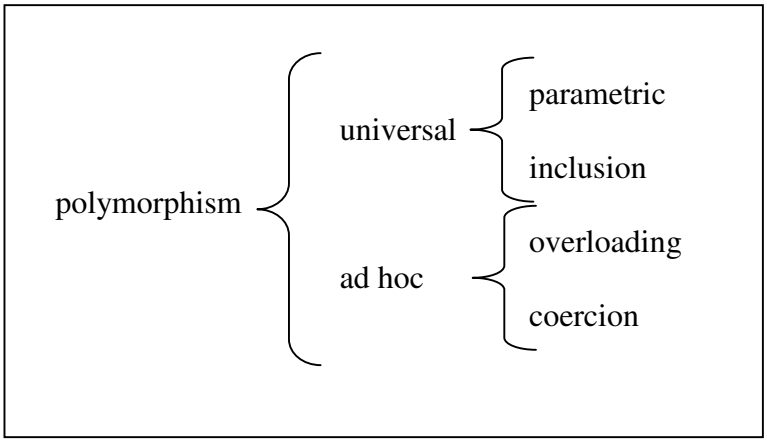

Figure 3.15. Kinds of polymorphism.

The two forms of ad hoc polymorphism are commonly found in many programming languages. *Overloading* means to provide many different implementations of the same function for different arguments. *Coercion* is an

operation provided to convert a type to another type to make it useful for the context.

The two forms of universal polymorphism supply code that is executed uniformly for a variety of types. Parametric polymorphism abstracts over types. It is obtained when a unit (function, class or module) works uniformly over a range of types [CW 85]. Types can be supplied as parameters to instantiate that unit.

Inclusion (subtype) polymorphism is essential to model some object-oriented programming features. In inclusion polymorphism an object can be seen as belonging to many different classes. In this context, an object can substitute another object when the former is a subtype of the later. Types and subtypes as well as the rules of the subtype relation are explained in section 3.8.

Object-oriented languages obviously favor subtype polymorphism. They tend to ignore parametric polymorphism as they can simulate it to a limited extent.

Not all object-oriented languages support all four forms of polymorphism. Two of the most popular ones, Java and C#, do not support parametric polymorphism. Instead they use subtype polymorphism to simulate it. However this technique is not type safe and runtime errors can arise. Subtype polymorphism cannot replace parametric polymorphism in a safe manner. Proposals to include parametric polymorphism in both languages have been analyzed recently [BCK+ 03, C# 02].

Chapter 4 presents several approaches of parametric polymorphism.

3.7. Types and subtypes.

Types are an important part in programming languages. Types are useful to represent abstractions, document programs, and detect errors at compile time. They allow the generation of efficient code and can help to provide runtime safety.

A type is a set of values and suitable operations on those values. The distinction of types in programming languages goes back to the 1950's when the first high level programming language i.e. FORTRAN, was developed. Some languages like Pascal, and Java, require explicit type annotations while others like SML use a type inference mechanism to discover the types of the elements.

A subtype is a type derived from another type. Subtypes are compatible with their base types and support all their operations. A subtype can be used in any context where an object of its supertype is expected. This notion is called "*subsumption*: an object can subsume another object that has a more limited protocol." [AC 96]. The basic rules of subtyping are subsumption, reflexivity, and transitivity.

The rule of subsumption is shown in figure 3.16 It states that if we have a expression *v* of type *S* (v:S) and we know that type *S* is a subtype of type *T* (S<:T) then *v* has also type *T* (v:T).

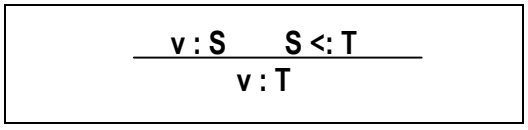

Figure 3.16. Rule of subsumption.

In general the subtype relation is defined as follows: assume *S* and *T* are types. Type *S* is a subtype of *T*, written as S <: T, iff the set of values of *S* is a subset of the set of values of *T*. That means that a value of type *S* can be safely used in contexts where a value of type *T* is expected.

Subtyping rules should be reflexive (S <: S) and transitive (S <: T and T <: U implies S <: U).

The type of a function contains two parts (D −> R). A function *S* is a subtype of a function *T* iff the domain of S is a supertype of the domain of *T* and the range of *S* is a subtype of the range of *T*. In this context a function of type *S* can be used in any context where a function of type *T* is expected. Figure 3.17 shows the subtype rule for functions, which shows a contravariant relation for the domain and covariant relation for the range.

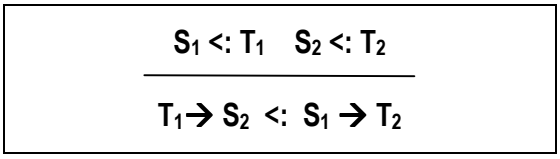

Figure 3.17. Subtype rule for functions.

A record *S* is a subtype of a record *T* if *S* has at least the same identical fields or more than *T*. This is known as *width subtyping* and the rule is shown at the left of figure 3.18. *Depth subtyping* allows variations in the type of corresponding fields if they are in a subtype relation. This rule is presented at the right of figure 3.18.

$\{a_i : T_i \; ^{i \in 1..n+k}\} <: \{a_i : T_i \; ^{i \in 1..n}\}$	$\dfrac{\text{for each } i \quad S_i <: T_i}{\{a_i : S_i \; ^{i \in 1..n}\} <: \{a_i : T_i \; ^{i \in 1..n}\}}$
Width subtyping	Depth subtyping

Figure 3.18. Subtype rules for records.

Many strongly-typed object-oriented languages like Eiffel [M 92], Java [GJSB 00], and C# [C# 01] do not separate the class hierarchy from the type hierarchy. They use inheritance to define the type hierarchy of objects, e.g., classes generate types and subclasses generate subtypes. Other languages, like Objective ML [RV 98], MOBY [FR 99], and PolyTOIL [BFSG 03] separate the class hierarchy from the type hierarchy using structural equivalence.

Cook, Hill, and Canning [CHC 90] recommend to separate the subclass relationship from the subtype relationship because when they are unified either the type system of the language is not safe, e.g., Eiffel [Co 89], or the language loses expressiveness since specialization of parameters when a method is overridden is not possible, e.g. Java. A well-known problem to illustrate this situation is the presence of binary methods in derived classes. Figure 3.19 shows a class definition with a binary method and a subclass definition. The left part of figure 3.19 is borrowed from [CHC 90] and uses Eiffel, while the right part shows the same problem written in Java.

```
// Eiffel
class P feature
  i : Integer is 5;
  id : Like Current is Current
  eq(other : Like Current) : Boolean is
   begin
      Result := (other.i = Current.i)
   end
end P

class C inherit P redefine eq feature
  b : Integer is 5;
  eq(other : Like Current) : Boolean is
   begin
      Result := (other.i = Current.i) and
                (other.b = Current.b)
   end
end C

p : P := new P();
c : C := new C();
v : P := c

v.eq(p);   // ERROR  in method eq of class C
```

```
// Java
class P {
  Integer i = new Integer(5);
  boolean eq(P p) {
    return (this.i == p.i);
  }
}

class C extends P {
  Integer b = new Integer(5);
  boolean eq(P p) {
    if (p instanceof C)
      return ((this.i == p.i) && (this.b == ((C)p).b));
    else return (this.i == p.i);
  }
}
```

Figure 3.19. Binary method problem in Eiffel and Java.

In Eiffel, the parameter of method *eq* is covariantly specialized. Three variables are defined (p, c, and v). The static type of variable *v* is *P* so *v.eq(p)* is well typed. But at runtime, the program executes method *eq* of class *C* because variable *v* contains an object of type *C*, which terminates the execution with an error because object *p* doesn't have a field *b*.

In the Java version, the method *eq* of class *C* cannot change its signature because it overrides the inherited method *eq*. The argument must be cast before field *b* is compared. The programmer has to deal with these implementation details.

3.8. Some other concepts.

3.8.1. Type equivalence.

Type compatibility needs to be defined to allow type checking. Two types are compatible if they are equivalent. There are two ways to define type equivalence: by name and by structure. In languages with name equivalence like Ada, two variables are compatible when they are defined with the same type name. The language must allow the definition of type names in order to support name equivalence. Languages with structural equivalence like Modula-3, allow two variables to have compatible types if the types used in their definition have the same structure. These two variations have their own advantages and disadvantages. Name equivalence is easier to implement than structural equivalence, but it is also more restrictive.

3.8.2. Typechecking.

A program can be typechecked statically at compile time, or dynamically at runtime. The main advantage of statically typed programming languages is that many errors can be detected at compile time. Dynamically typed languages are more flexible but the program can fail due to type errors at runtime.

Some features of object-oriented languages like subtyping and inheritance make it difficult to typecheck these languages. The type system of the language

must specify a set of rules that allow to typechecking programs and detect type errors.

Types can change in the type hierarchy in two ways: covariantly and contravariantly. Types change in a covariant way when they are parallel to the type hierarchy and in a contravariant way when they change in an opposite way to the type hierarchy. They are called invariant when they do not change at all. Covariance and contravariance characterize two different relations: specialization and subtype respectively [C 95].

The addition of new members in a subclass does not pose a problem to the type system. However, overriding a method in a subclass can generate a problem if the return type does not change in a covariant way or the type of the parameters does not change in a contravariant way, from the method being overridden. Another kind of problem arises when the subclass relation is used to create the subtype relation. These two problems and the solutions offered by several languages are presented in [BCC+ 96].

Type safety is preserved in subclasses allowing covariant specialization for return types, contravariant for parameters and invariant for instance variables [B 02].

Chapter 4.

Generics.

Parametric polymorphism is a kind of universal polymorphism where the same code is used for several types. Parametric polymorphism is an important feature that increases language expressivity and clarity, and improves program safety. In some programming languages this feature is called a generic. Generic programming is the ability to write code once and reuse it in different circumstances. Generic code defines an abstraction independently of the data types to be used at runtime.

This chapter presents different kinds of genericity found in programming languages. Different approaches to translation are described. Examples using generics in different programming languages are presented.

4.1. Parametric polymorphism.

The basic idea of genericity is "substitution of type annotations" [PS 94]. Parametric polymorphism allows the definition of code that works uniformly for different types, which can be unrelated. This form of implicit universal parametric polymorphism is frequently found in functional languages like ML and Haskell. Figure 4.1 shows two functions, written in Objective Caml (OCaml), that work for any type.

```
# let identity x = x;;
val identity :  'a  ->  'a = <fun>

# let head = function
  [] -> failwith "Empty list"
  | h::t -> h;;
val head : 'a list -> 'a = <fun>

# head [1;2;3;4];;
- : int = 1

# head [[1.0; 1.3; 1.5];[2.0];[3.0]];;
- : float list = [1. ;1.3 ; 1.5]

# head [["list1";"a"];["list2";"b"];["list3"]];;
- : string list = ["list1";"a"]

# head [];;
Exception: Failure "Empty list".
```

Figure 4.1 Two Polymorphic functions in Objective Caml.

Function *head* receives as a parameter a list of any type of elements and returns as a result the first element of that list. The function is polymorphic and can be called to execution sending as argument a list of any type. In the example, the

function *head* is called three times; the first one with a list of integer values, the second one with a list of list of float values and the third one with a list of list of string values and the last one with an empty list.

The addition of parametric polymorphism in statically type-checked programming languages enhances the expressivity of the language, reduces code maintenance, and increases safety because more errors are detected at compile time.

4.2. Kinds of genericity.

Generic code can be unconstrained or constrained. Unconstrained genericity means that the parameterized type would receive any type available in the system to create an instance. On the other hand, constrained genericity means that the parameterized types would receive as parameters only those types that agree with the restrictions imposed.

We introduce some concepts that are needed in this section.

- A *type variable* is a name that stands for an indeterminate type.
- A *parameterized type* is a type that depends on one or more other types.
- *Covariance.* The type of an element in a class is replaced with a subtype in a derived class, i.e., changes of a particular type are parallel to the type hierarchy.

- *Invariance*. The type of an element of a class does not change in a derived class.
- *Contravariance*. The type of an element of a class, is replaced with a supertype in a derived class i.e., changes of a particular type are opposite to the type hierarchy.

4.2.1. Unconstrained genericity.

In unconstrained genericity, any type in the system can be used as type parameter in the instantiation of a parameterized type. There is no restriction. In this section we use the language Eiffel to show an example of unconstrained genericity. Eiffel defines a generic class as a class that accepts type parameterization. Figure 4.2 shows the definition of a parameterized class in Eiffel.

```
class STACK [T]
  feature
    store : array[T];
    size: Integer := 0;
    push( elem : T ) is
    do
        store( size ) := elem;
        size := size + 1;
    end -- push
    pop : T is
    do
      result := store(size);
      size:= size - 1;
    end --pop
-- STACK
```

Figure 4.2. Parameterized class STACK written in Eiffel.

The type parameter *T* defined in the class has no bounds or restrictions imposed. That means that any type can be used as type parameter to create an instance of class STACK. An instantiation of the generic class will create a specific instance of the generic class. Two instantiations of class STACK are shown in figure 4.3.

The result of these instantiations is as if two versions of the STACK class had been written, one for each type.

```
pointStack : STACK[Point];
integerStack: STACK[Integer];
```

Figure 4.3. Two instantiations of class STACK.

4.2.2. Constrained genericity.

There are several mechanisms to restrict the type variables used in instantiations of parameterized types. The one proposed by Cardelli and Wegner [CW 85] called system $F_{\leq}$, allows expressing the idea that a function can be applied to all types that are a subtype of another. This mechanism is not powerful enough to express all kind of constraints like the ones needed for recursive type definition. Other more powerful mechanisms that are able to express constraints with recursive type definitions are *F-bounded quantification* [CCH+ 89], *where-clauses* [DGLM 95], and *matching* [BFSG 03].

F-Bounded is a type system that generalizes system $F_{\leq}$ with subtyping to model basic features of object-oriented languages. In this system the bound of a quantified type can depend on itself. Where clauses were proposed as an alternative to subtyping where the type constraints are specified explicitly by listing the required methods (name and signature) for the parameters. Matching requires the separation of types and classes. Matching is a relation between types that generalizes subtyping, i.e., it is less restrictive.

Modula-3 supports the definition of generic interfaces and modules and uses interfaces to bound formal parameters to actual interfaces when the generic unit is instantiated [N 91]. Different approaches to define constraints like *virtual types* and *where-clauses* are used in the languages BETA and Theta respectively [MMN 93, DGLM 95].

Examples of simple and recursive type constrains are presented in sections 4.2.2.1 and 4.2.2.2.

4.2.2.1. Simply bounded genericity.

This approach restricts the type of the parameters used in the instantiations of parameterized types. The parameter type must be bounded to another type to ensure that it implements some needed methods.

GJ is based on F-Bounded quantification [BOSW 98], but in this section we do not use the recursive type constraint to show the problems that can arise when

simple bounds are defined. An example of a parameterized class in GJ is shown in figure 4.4.

```
interface Orderable {
   boolean eq (Orderable other);
   boolean le (Orderable other);
}

class OrderedList< T implements Orderable> {
   T listElem;
   …
   public void insert (T elem) {
      …
      if (listElem.le(elem))…
      …
   }
   public boolean member (T elem) {
      …
      if (listElem.eq(elem)) return true;
     …
   }
}
```

Figure 4.4. A parameterized class with a simple bound in GJ.

OrderedList is a parameterized class that depends on a type parameter called *T*, which is bound to *Orderable*. This bound restrict the types that can be used to create instances of class *OrderedList*. Figure 4.5 shows some instantiations of the parameterized class *OrderedList*.

```
class Point implements Orderable {
  int x,y;
  boolean eq (Orderable other) {  … }
  boolean le (Orderable other) {  … }
  …
  public static void main (String [] args) {
    OrderedList<Point> pointList = new OrderedList<Point>();
     Point p1 = new Point(1,1);
    …
    pointList.insert(new Point(1,2));
    ...
    if (pointList.member(p1))  ...

  }
}
```

Figure 4.5. Instance creation of the parameterized class *OrderedList*.

The class *Point* implements the interface *Orderable* and it can be used as type parameter to create an instance of the class *OrderedList*. An instance of the generic class *OrderedList* is created when a variable of type *OrderedList<Point>* is created.

The definition of parameterized types with simple bounds like the one used in the previous example cause some problems when binary methods are needed in the class. For example, the interface *Orderable* contains two methods, each one with a formal parameter of type *Orderable*. Classes that implement this interface are not allowed to covariantly change the type[1] to specialize it due to type safe restrictions. At runtime, a variable of any type that implements *Orderable* could be

[1] The language Eiffel allows covariant changes of parameters, but it's been proven that its type system is unsound. [Co 89]

passed as argument and the compiler is unable to detect that error. The programmer is responsible to write special code to ensure that the correct type is received and to cast it to the type needed to execute the operations. This limitation can be solved using a generalized form of parametric polymorphism with recursive bounds, which is presented in next section.

4.2.2.2. Recursively bounded genericity.

It is possible to define type parameters with recursive constraints using *F-bounded quantification* [CCH+ 89]. Using a recursive bound on the type parameter ensures that the required arguments have the same type than the object that receives the message, but they may not be exactly the same.

Generic Java (GJ) relies on F-Bounded quantification to allow the definition of parameterized types [BOSW 98].

Figure 4.6 shows the definition of a generic interface *Orderable* and the generic class *OrderedList* using a recursive bound in the type parameter.

```
interface Orderable <T>{
   boolean eq (T other);
   boolean le (T other);}

class OrderedList< T implements Orderable<T>> {
  // same as before
}
```

Figure 4.6. A parameterized class with a recursive bound in GJ.

When the interface *Orderable* is parameterized, it is possible to create recursive bounds in the type parameter of a class bounded to *Orderable*. In *OrderedList<T implements Orderable<T>>* the type parameter *T* is constrained to implement an interface that is parameterized with itself. A type parameter is needed to define an instance of *OrderedList* class. The type that can be used to instantiate *OrderedList* is restricted to implement the parameterized interface *Orderable* with itself. Figure 4.7 shows an example of a class that instantiate the *OrderedList* class.

```
class Point implements Orderable <Point> {
  int x,y;
  boolean eq (Point other) {  … } //type specialized to Point
  boolean le (Point other) {  … }  //type specialized to Point
  …

  public static void main (String [] args) {
    OrderedList<Point> pointList = new OrderedList<Point>();
     Point p1 = new Point(1,1);
    …
    pointList.insert(new Point(1,2));
    ...
    if (pointList.member(p1))  ...

  }
}
```

Figure 4.7. Innstantiation of the parameterized class *OrderedList*.

Class *Point* can be used as an actual type parameter to create instances of *OrderedList* because *Point* implements *Orderable<Point>*. We instantiate the class as follows: OrderedList<Point> pointlist = new OrderdedList<Point>( );

In GJ, when a class is defined with a recursive bound the possibility of deriving new classes that can be used as type parameters is lost. F-Bounded quantification and binary methods cannot be smoothly combined in languages with nominal subtyping, like Java. We illustrate that with the example shown in figure 4.8.

```
class ColorPoint extends Point {…}

OrderedList<ColorPoint> cpList = new OrderdedList<ColorPoint>;
                    // compile-time error!! class ColorPoint does not implement Ordeable<ColorPoint>

// another definition
class ColorP extends Point implements Orderable<ColorP> { … }
// compile-time error!!   class ColorP inherited Orderable<Point> and cannot implement Orderable<ColorP>
```

Figure 4.8. Subclasses cannot be used as type parameters.

The definition of class *ColorPoint*, generates a class that cannot be used as type parameter of *OrderedList* because it inherits *Orderable<Point>* and it needs *Orderable<ColorPoint>*. The compiler produces an error.

The definition of class *ColorP* is not valid either due to a restriction imposed by the technique used in the implementation of parameterized types in GJ. Class *ColorP* inherits *Orderable<Point>* and cannot at the same time implement *Orderable<ColorP>*. The explanation is found in [BCK+ 01].

> To support translation by type erasure, we impose the restriction that a class or type variable may not at the same time be a subtype of two interface types which are different parameterizations of the same interface. Hence, every superclass and implemented interface of a parameterized type or type variable can be augmented by parameterization to exactly one supertype.

4.3. Translation.

Parametric polymorphism in programming languages can be implemented in three different ways:

- *Heterogeneous code*. Specializing the code for each instantiation.
- *Homogeneous code.* Generating common code for all instantiations.
- *Hybrid code*. A combination of homogeneous and heterogeneous code.

Each translation approach has its own advantages and disadvantages. Instances of parameterized classes might be created at compile time, link time or execution time. Ada and C++ use the heterogeneous translation that produces a specialized version of the code for each different instance of the parameterized type at compile time. The advantages of heterogeneous translation are that any type can be used as actual parameter to create an instance and no runtime cost penalties exist because efficient code is produced. Some disadvantages of heterogeneous translation are: compilation is slower, the source code is needed at compile time when an instance is defined and it can cause great memory consumption at runtime when many different instances exist. On the other hand languages like Modula-3 and ML follow the homogenous translation where a single block of code is generated to manipulate all possible different instances. At runtime, there is only one block of code that is shared by all instances, but the execution performance can be affected by the extra indirection through references.

4.3.1. Homogeneous.

A homogeneous translation produces a single piece of code that works uniformly for all types. The implementation of generic code in GJ has followed the homogeneous approach [BOSW 98b]. A technique called type erasure is used to translate the generic code that can be executed by the Java Virtual Machine (JVM) [BOSW 98]. The translation technique can be described in four steps:

1. Erase type parameters
2. Replace type variables with their bounding type
3. Add cast operations
4. Insert bridge methods.

In this section we explain how this technique is implemented using a source code example.

The compiler translates a parameterized class into a class that replaces type parameters with their bounding types, generally Object, which is the type at the top of the class hierarchy. This is why it is called type erasure; their bound types replace all type parameters. Sometimes *bridge* methods are needed to ensure that overriding works properly. Classes that use instances of a parameterized type require the insertion of some cast operations where methods that return the type parameter are called to execution. This cast operations inserted by the compiler are warranted not to fail at execution time. The resulting class is similar to a class implementing the generic idiom.

Figure 4.9 shows the translation of a parameterized class using source code. The class in the top left of the figure, *Stack*, is a parameterized class with one type parameter, *<T>,* that is used to define the type of some elements of the class. The class at the top right of the figure, *TestStack,* is used to create an instance of the parameterized class *Stack<T>* using *String* as the actual type parameter. When the *pop* method is invoked no cast operation is needed because the type of the object returned by *pop* is the one expected (String).

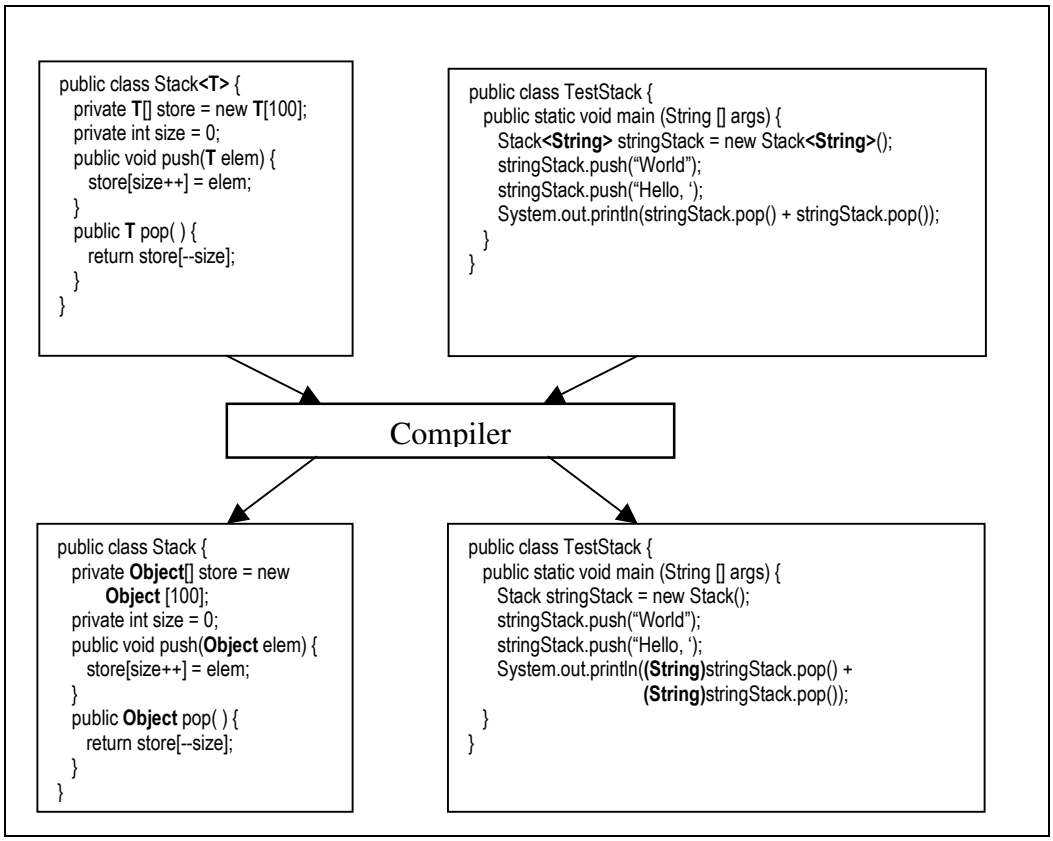

Figure 4.9. Translation of a parameterized class.

These two classes are translated to the classes listed in the bottom part of the figure. All type parameters are erased from class *Stack*<T> and elements annotated as *T* in *Stack*<*T*> are now annotated as *Object* in the resulting class *Stack*. The class *TestStack* refers to the class *Stack*, without type parameters, and cast operations are inserted when the *pop* method is invoked.

This translation technique does not allow using primitive types as type parameters to create instances of parameterized types. They cannot be used as type parameters because they cannot all be unified with a particular type that will allow them to be treated uniformly. A different technique could be implemented to support primitive types as type parameters but some changes to the JVM are required. In order to preserve compatibility with legacy code, changes to the JVM were avoided.

PolyJ [MBL 97] is another proposal to include parameterized types in Java. PolyJ allows the use of primitive types as type parameters, uses where-clauses to define constrains, and implements a homogeneous translation approach. However, they make changes to the JVM in order to produce a more efficient translation.

4.3.2. Heterogeneous.

A heterogeneous translation produces a piece of code for every different instantiation of the parameterized type. The only difference of each piece of code is the type of the elements they contain. They define the same behavior for different

types. C++ uses a heterogeneous translation approach. A template class is defined with some type parameters. There is no way to constrain the type parameters so type checking is done at linking time. For every type instantiation, the compiler will generate a specialization of the class. Figure 4.10 shows an example of a template and two instantiations of it.

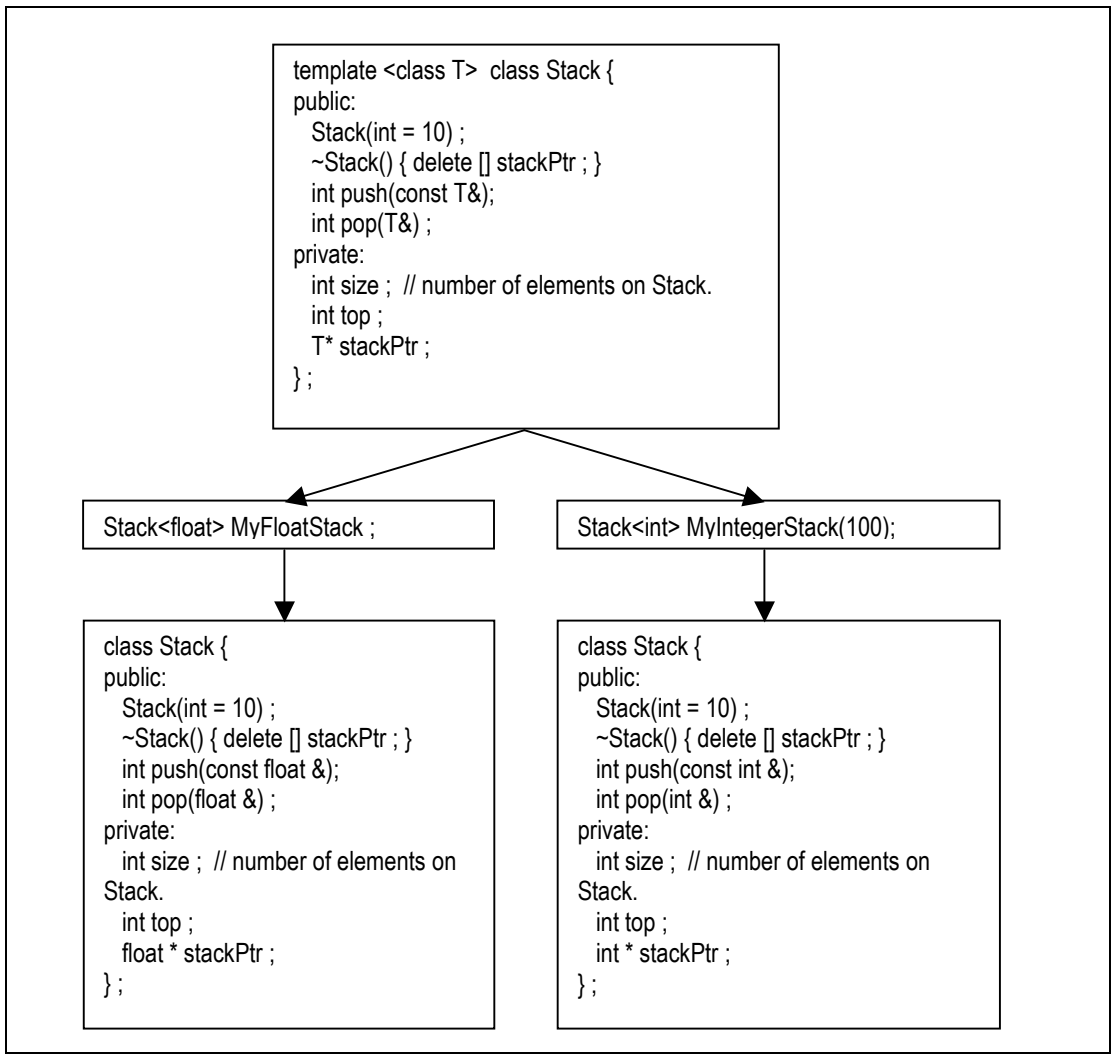

Figure 4.10. A template class with two instantiations.

This approach generates efficient code because every instantiation produces a specialized piece of code. On the other hand, if many different instantiations are needed, the same code for different types is produced and the program size grows.

A proposal for adding parameterized types to Java that includes the use of primitive types as type parameters, uses a heterogeneous translation approach where the specialization of code is generated at loading time [AFM 97].

4.3.3. Hybrid.

A hybrid translation may produce both heterogeneous and homogeneous code. The programmer can choose the translation mechanism, as in Pizza, or the compiler can decide which translation is more appropriate for each instantiation.

The C# implementation of generics uses both translation approaches: code specialization and code sharing [KS 01]. Instantiations of parameterized classes are loaded dynamically and the code of their methods is generated on demand. They generate unshared code for primitive instantiations and possible-shared code for the rest. This distinction is transparent for the programmer because C#'s type system is unified. Although value types are stored in a different way than reference types, C# provides automatic boxing and automatic unboxing of values to avoid explicit wrapping.

Figure 4.11 shows an example of a parameterized class and two different instantiations.

C# is translated to the Intermediate Language (IL) of the Common Language Runtime (CLR). The extension of the CLR to support generics proposed by Kennedy and Syme [KS 01], has three main points: adding some new types to the IL type system, introducing polymorphic forms of the IL declaration for classes, interfaces, structs, and methods along with ways of referencing them, and specifying some new instructions and generalization of existing instructions. This extension of the CLR allows an efficient execution of generic code not only for C# but for all programming languages supported by the .NET platform. A formalization of this mechanism is presented in [YKS 04].

```
public class GStack<T> {
  private T[] store = new T[10];
  private int size = 0;
  public void Push(T elem) {
    if (size >= store.Length) {
      T[] tmp = new T[size*2];
      Array.Copy(store, tmp, size);
      store = tmp;
    }
    store[size++] = elem;
  }
  public T Pop() {   return store[--size];   }

  public static void Main () {
    GStack <int> intStack = new GStack<int>();   // specialized code for this instantiation
    intStack.Push(17);
    int y = intStack.Pop();

    GStack <string> stringStack = new GStack<string>();  // shared code for this instance
    stringStack.Push("hello");
    string s = stringStack.Pop();
  }
}
```

Figure 4.11. A parameterized class with two instantiations in C#.

The new features of the Java programming language include automatic boxing and unboxing [BG 03]. This inclusion will automatically box and unbox primitive values into reference types, making source code more readable.

4.4. Examples of generics in some PL.

A genericity mechanism is a language construct that allows the definition of generic programs. There are some distinct mechanisms that have been adopted by different languages. In this section we explore some of them.

4.4.1. Templates in C++.

A template is a pattern used to create multiple instances of something. Templates are the genericity mechanism supported by C++. A C++ template defines a family of types or functions [S 91]. Templates are similar to definitions of parameterized types. A parameterized type contains one or more type parameters, which are used to create instances of it. Figure 4.12 shows a stack template definition written in C++ as well as some instantiations of it. The complete implementation of the template is omitted.

```
template <class T>  class Stack {
public:
   Stack(int size) ;

   void push(const T&);
   T& pop() ;
private:
   int size ;  // number of elements on Stack.
   int top ;
   T* stackPtr ;
} ;

Stack<float> MyFloatStack ;
Stack<int> MyIntegerStack(100);
```

Figure 4.12. Stack template in C++.

C++ implements templates by an approach called macro expansion. For every instantiation of the template a specialized class is generated replacing the type parameter of the template for the actual type of the instantiation. In the example presented in figure 4.12 two instantiations of the template Stack are made and two Stack classes are generated, one for type *float* and another one for type *int*. Type checking is performed only on the function instance, not on the template itself. Using templates for generic types have some advantages and disadvantages. The main advantage of templates is that they allow an efficient implementation because code is specialized for every type. However, this may cause 'code bloat', since for every combination of parameter values that is passed to the generic type a new instance of the template is created. However, this technique is attractive if we are concerned with the performance of the code generated.

4.4.2. Parameterized classes in Pizza.

A parameterized type defines a group of related types that have similar behavior but differ in the types they manipulate. A parameterized type is a type definition with a list of type parameters. Eiffel uses this mechanism to define parameterized classes. Many proposals to extend the Java programming language with parameterized types have been made [BOSW 98, CS 98, AFM 97, EKMS 97, MBL 97]. Pizza [OW 97] is a superset of Java that includes parametric polymorphism, higher-order functions, and algebraic data types. Figure 4.13 shows a parameterized class in Pizza.

```
public class Stack<T>
  private T[] store = new T[100];
  private int size = 0;
  public void push(T elem) {
    store[size++] = elem;
  }
  public T pop( ) {
    return store[--size];
  }
}
```

Figure 4.13. A definition of a parameterized class in Pizza.

Stack is a parameterized class that has a type parameter called *T*. In this case, no constrains are defined for the parameter. Any type can be used to instantiate the *Stack* class. Pizza allows the programmer to define the translation approach to generate code. It can be a generic class -the same code for all instances,

or a specialized class -different code for each different type instantiation. Figure 4.14 shows an example where the parameterized class *Stack* is instantiated.

```
public class TestStack {
   public static void main (String [] args) {
      Stack<String> stringStack = new Stack<String>();
      stringStack.push("World");
      stringStack.push("Hello, ');
      System.out.println(stringStack.pop() + stringStack.pop());
   }
}
```

Figure 4.14. Instantiation of the parameterized class Stack.

As consequence of the use of parameterized types, two kinds of classes exist; those with type parameters and those without type parameters. In languages with nominal subtyping, these two kinds of classes differ not only by having type parameters but also because the subtype relationship does not extend to instances of parameterized classes, e.g., an instance of a generic class is not a subtype of the generic class. Another problem present in Java is that instances of parameterized classes with binary methods inhibit the use of inheritance to derive new classes that can be used to create instances of a parameterized class [BS 03b].

4.4.3. Virtual binding in BETA.

The BETA language supports a genericity mechanism called virtual binding [MMN 93]. Virtual binding is expressed in BETA with virtual patterns that contains a virtual attribute with a virtual bound.

A virtual pattern in BETA is similar to a parameterized class in other languages but the type parameters are virtual attributes with bounds. When an instance of the virtual pattern is created a final bound for the virtual attribute must be defined. Figure 4.15 shows the definition of Stack in BETA.

```
Stack: (#
  T:< Object;            {virtual attribute with virtual bound }
  store: [100] @ T;
  top:@ Integer;
  Init: (# do 0 -> top #);
  Push: (#
    x:@ T;
    enter x;
    do top+1 -> top; x-> store [top];
  #);
  Pop: (#
    do top-1 -> top;
    exit store [top+1];
  #);
#);
```

Figure 4.15. Generic Stack in BETA.

The definition of *Stack* contains a virtual attribute *T*, which is the generic parameter that is virtually bound to *Object*. Subpatterns can make further restrictions by defining new bindings of the virtual attribute. An instance of the virtual pattern is created when a final binding of the virtual attribute is done.

```
intStack : Stack (# T:: Integer #);
```

New classes can be constructed by extending the binding, but the extended binding must be any subpattern of its previous bound.

```
es: ElementStack : Stack (# T ::< Element #);
```

In this approach, there is only one type of class and the bindings can be extended many times. However, the main disadvantage of this approach is that it fails to preserve static type correctness [PS 94].

4.4.4. Class substitution in BOPL.

Class substitution is another approach to genericity proposed by Palsberg and Schwartzbach [PS 94]. This approach was developed as a complement to inheritance. It is a new subclassing construct where classes derived using inheritance cannot be derived from generic classes and vice versa.

The construct appears in BOPL [PS 94], a simple object-oriented language. Figure 4.16 shows a partial definition of Stack in BOPL and how subclasses are obtained using class substitution.

```
class Stack
   var store : Array[Object <- Elem];
   var elem :Elem;
   var top : Int;
   method push ( other : Elem)
   …store.atput(top, other);
       top := top +1;
      self
   end…
   method pop ( ) returns Elem
   …
   end
end

class IntStack is Stack[Elem<- Int];
class BoolStack is Stack[Elem <- Bool]
```

Figure 4.16. Stack definition and instantiation in BOPL.

A new class *IntStack* is derived from the class *Stack* by replacing the formal type parameter *Elem* by the type Int. Similarly for *BoolStack*.

Chapter 5.

Analysis and Goals.

Object-oriented technologies have proved their applicability in several areas of software development. They are widely used in the analysis and design of systems and programs.

Over the past two decades, existing programming languages like Ada, C, and even COBOL among many others have embraced object-oriented features. Examples of this evolution are Ada 95, C++, Objective C, Objective COBOL, CLOS, and Objective ML.

The fact that many languages have evolved to include object-oriented features motivates us to start the design of the language by including objects. Imperative languages with extensions for objects allow program development using a combination of procedural and object-oriented features.

The design of a programming language is often directed by requirements to fulfill some needs. Simplicity, expressivity, powerfulness, and elegance are some attributes considered in the design process.

In this chapter we explore language design keeping modularity, genericity and objects as the main features of the language. We describe the desired features of a language with simplicity as the most valuable attribute. We describe some problems and analyze the approach to solve them in some object-oriented languages that neither support modules nor parametric polymorphism. Finally we describe the goals of our language design to combine these constructs.

5.1. Core features.

The design of a complete language involves many decisions. There are many languages designed as descendents of other languages because they include some of their features. We have chosen to follow this path, selecting some elements of several languages to start with. This decision will allow programmers to become familiar with our language easily.

We took basic elements of Java, C#, and Modula-3. For example, we adopted the block enclosing construct **{ }** of Java and C# avoiding the more verbose **BEGIN END** of Modula-3. We reduced the number of basic types, literals, and operators to create expressions. The number of statements was also minimized to allow expressing basic computations. We selected a minimal core language because these features are not the thrust of this thesis.

5.2. Classes or objects?

The main concept in object-oriented languages is the use of dynamic entities that carry with themselves their own data and functionality and interact with other entities through message passing. These entities are called objects. Two different ways to define and create objects are supported by the two models of object-oriented languages: passive and active languages.

In some passive object-oriented languages the class construct establishes a syntactic scope for its elements, but some languages, like C++, allow the partial elimination of these boundaries. A class in C++ declares as *friends* all functions or classes that are allowed to have access to its elements. This strategy requires deciding in advance which other classes or functions are going to have access to the elements of a class. If a new class needs to be declared *friend* the source code needs to be updated. C++ violates encapsulation principles because access to elements of classes is allowed outside the class definition.

Frequently object-oriented languages with classes contain a nominal type system. In these languages the class mechanism poses a major problem when binary methods are part of a class and a derived class needs to override them to specialize their behavior. The solution to this problem is not trivial if type safety must be preserved. Either the type hierarchy must be decoupled from the class hierarchy adding complexity to the language or the expressiveness of the language

is limited leaving the programmer responsible of the correct implementation of binary methods.

It is commonly recognized that classless languages are simpler than languages with classes because they have only one mechanism to define and generate objects. Most of these classless languages are dynamically typed or offer only some static type checking and no guarantee that programs won't crash at runtime due to type errors. The main advantage of these languages is that they can be used for exploratory programming to rapidly prototype applications. Prototypes enhance the flexibility of the language by allowing the object to change its behavior at runtime.

Lieberman, Stein, and Ungar agreed in "*The Treaty of Orlando*" [LSU 87] that neither model is the best for all situations.

Ungar said "*that no new languages should be designed with classes*" [U 88], because classless languages are simpler and more expressive. At the same time, Lieberman [L 88] recognized that these two sharing mechanisms (inheritance by classes and delegation) have different application areas.

The complexity of large systems seems to be better modeled with class-based languages that provide static type checking. Wadler [W 87] expresses this in the following paragraph.

> However, when a prototypical system or untyped formalism is used to model a complex universe, types and classes for expressing regularities in the domain creep in by the back door, and it become preferable to introduce explicit typing and classification schemes rather than rely on ad hoc ingenuity.

Regardless of the claim that object-based languages are simpler and more expressive than class-based languages, we believe that they are not good enough to express abstractions with a guarantee of an identical and predictable behavior for all its instances. Furthermore their dynamic typing fails to provide a language to develop safe and large programs that can be easily maintained.

A goal in our design is to provide a construct to define objects incrementally facilitating code reuse. All objects created from the same entity must provide an identical behavior at runtime and they can be typecheked at compile time. This mechanism must be simple, should not violate encapsulation, and avoid overlapping with other mechanism in the language.

5.3. Type annotations and typechecking.

Since the first high-level language was designed, types have been an essential part of programming languages. They have several uses in programs. Types delimit the set of values a variable can hold. They serve also as documentation of a program and the compiler uses them to detect type errors and to generate more efficient code.

Explicitly typed languages are those for which type annotations are used in programs as opposed to implicitly typed languages where no type annotations appear in programs and the compiler infers this information in order to check type

consistencies. Type annotations help the programmer to document the source code of programs.

Programs can be checked statically at compile time, dynamically at runtime, or a combination of both. These two approaches have advantages and disadvantages. Statically typed languages are more restrictive than dynamically typed ones. Yet static type checking allows detecting errors at compile time while dynamic type checking detects errors at runtime. Dynamically typed languages are more flexible, accepting more programs, but they spend time at runtime to do the typechecking. Statically typed languages seem to be better for the development of programs because more errors can be detected at compile time and no runtime checking is needed.

Strongly typed languages offer a guarantee that no runtime type errors are possible. "*A safe language is one that protects its own abstractions*" [P 02].

Our goal is to provide an explicitly typed object-oriented language that can be used to define new types and in which programs can be type checked statically.

5.4. Subtypes.

Subtype polymorphism is fundamental in object-oriented languages. This feature allows defining a relationship on types such that the objects of a subtype can be seen as objects of their supertypes. Subtypes can be used safely in any

context where a supertype is expected because they provide the same elements as their supertypes and perhaps more.

Object-oriented languages define the subtype relation in different ways. In languages with nominal subtyping, the inheritance hierarchy defines the type hierarchy, i.e. in Java subclasses define subtypes. This approach leads to impose restrictions in derived classes in order to preserve type safety, i.e. the type parameters of methods cannot change covariantly. On the other hand, C++ explicitly separates inheritance and subtyping [S 91]. When a class is derived with private inheritance, it inherits the implementation but not the structure of its parent, while a class derived with public inheritance, inherits both implementation and structure from its parent.

In languages with structural subtyping, like OCaml and MOBY, the hierarchy of classes does not necessarily coincide with the hierarchy of types. In OCaml, the subtype relation must be explicitly annotated [CMP 00]. In MOBY structural subtyping is defined for object types and nominal subtyping for class types. Modula-3 is another language with structural equivalence, but an object type can be annotated with a "brand" which means that its resulting type will be different from any other type but the subtype relation is based on the inheritance hierarchy.

Explicitly typed languages must define the type of objects separately from the class definition. They provide more information improving readability but at

the same time they tend to be verbose. Implicitly typed languages do not suffer this problem because the compiler infers the types. Separating the class hierarchy from the type hierarchy introduces some complexity to the language. An example of this complexity can be seen in figure 5.1.

The example presented in figure 5.1 is borrowed from [BFSG 03]. It describes a definition of a simple class called *HelloClass*. The program contains two types declarations *HelloClassType* and *HelloType* for the class and the instances respectively. It also contain the implementation of the class *HelloClass*. At the end, a variable *myMood* is declared, an instance of class *HelloClass* is created and assigned to it; finally two methods are invoked.

This program does not contain a subclass declaration, but we can see that defining the type of the class and the object requires a lot of code.

```
program easy;
type
  HelloClassType = ClassType (
      { happy : bool },                -- types of instance variables
      { setMood : bool -> void ;    -- types of methods
        printMood : void -> void });

  HelloType = ObjectType        -- method types only
      { setMood : bool -> void ;
        printMood : void -> void };

const
  HelloClass = class
    var
      happy = true : bool;
    methods
      setMood = procedure (theMethod : bool)
        begin happy := theMood; end
      printMood = procedure ()
        begin
          if (happy) then
            printString("Have a wonderful day!");
            newline(1);
          else
            printString("Go away!!");
          end;
        end;
  end: HelloClass;

var myMood : HelloType;
begin
  myMood := new(HelloClass);
  myMood.printMood();
  myMood.setMood(false);
end
```

Figure 5.1. An example of a simple class in PolyTOIL.

Structural equivalence is frequently used to define the subtype relation but sometimes it is the programmer's responsibility to define explicitly this relation, as in OCaml. These languages do not suffer any problem when derived classes specialize covariantly the arguments in binary methods, because subclasses do not

generate subtypes. On the other hand, languages with nominal subtyping do not require specifying the type of objects separately from the class. They use the class name to represent types. Matching is another relation on types that is not as strong as the subtype relation [BPF 97]. It is an alternative relation to express type compatibility and provide a safe implementation of classes with binary methods.

Some object-oriented languages provide interfaces as a language construct to define some features of classes. In some languages interfaces represent the types of objects. Objects generated from classes that implement several interfaces have all the types they represent and can be used in any context that type is expected.

Interfaces are useful to define types and can be used to some extent to model multiple inheritance. They may hide information listing only the elements that are available for clients. Different classes can supply several implementations of interfaces.

In order to keep the language simple we do not separate classes from types but we need to include class interfaces to describe types and hide information. Every class must implement an interface that describes the elements that are available to clients.

5.5. Inheritance.

Inheritance is a language mechanism that allows the definition of new entities based on existing ones. All object-oriented languages support some form of

inheritance. Class-based languages use classes to inherit the features of a parent class creating new classes that can use the code of the parent or redefine it to specialize its behavior. Object-based languages support inheritance through a mechanism called delegation, where the new object delegates some of its functionality to another one. Objects can change their functionality at execution time and these changes can affect other objects.

Sometimes a class can be defined using one or more parents. In languages with single inheritance only one entity acts as parent. When more than one entity is used as a parent the language supports multiple inheritance. The debate about the necessity to provide multiple inheritance in a programming language has been going on in the literature for years. The advantage of multiple inheritance over single inheritance is in being able to model more complex structures.

Entities can inherit structure using interfaces and implementation using classes or delegation. Entities that inherit structure must provide implementation before objects can be created. Multiple interfaces can serve as parents without problem because only one implementation will be provided. The disadvantage of multiple interface inheritance is that it is not possible to reuse code and every time an interface is inherited an implementation for it must be provided.

Languages that allow multiple inheritance of implementation must provide explicit solutions to the problems that can arise, i.e., name ambiguity and redundant method calls. C++ solution relies on the programmer to solve these conflicts by

providing explicit delegation as shown in the example of section 3.5.2. Other languages like Eiffel [M 92] provides a *rename* clause that the programmer can use to solve name repetitions and an internal solution linearizing the conflicting methods based on the class hierarchy and calling the first one in the sequence. The problem is that the method selected is not always the one intended by the programmer. None of the solutions provided by languages with multiple inheritance of implementation seems perfectly satisfactory; the problems generated with its use outweigh the benefits.

Another inheritance mechanism called mixins had been analyzed recently in proposals to extend existing languages [ABC 03, ALZ 03, B 92, BPV 98, FKF 98, P 01]. Mixins promises the benefits of multiple inheritance while avoiding its difficulties, but some new concepts are needed in the language to support mixins. In order to maintain simplicity, we do not consider including mixins in our language.

Our goal with respect to inheritance is to incorporate a simple inheritance mechanism that can be easily understood by the programmer. It must enable the derivation of new classes as specializations of existing ones, allowing code reuse.

5.6. Bindings.

Static and dynamic bindings are allowed in most object-oriented languages. Static binding takes place at compile time and dynamic binding at runtime. Calls to

procedures and functions that are statically allocated can be bound at compile. Objects offer the possibility of dynamically selecting the method to be executed. The incremental definition of objects makes possible reusing code previously defined in some parent class. This code can be overridden to specialize the behavior of objects. Special variables *super* and *this* can be used to invoke methods retaining the binding to the parent or to the actual object independently of where they are used.

The convention used in several object-oriented languages to define static or dynamic method invocation is different. Some languages like C++ and C#, define all methods to be static by default and require that dynamically located methods to be declared as *virtual*. Other languages like Java, define all methods to be dynamically located by default and require others to be declared explicitly as *static*.

Single or multiple dispatch defines the selection of the method to execute, designating an object to receive the message or implementing methods as a set of generic overloaded functions where no object is the receiver. Multimethods languages suffer encapsulation problems or selecting the best function when the types of the parameters do not match exactly with the type of the arguments [BC 97, CL 97, CLCM 00, CL 94, CM 99].

The dynamic existence of objects suggests that all their methods must be dynamically bound. Classes define behavior of objects, which is specialized in subclasses so the redefinition of methods fits more naturally with dynamic binding.

Static binding should be used for those procedures or functions that are not defined as part of classes.

5.7. OOL without modules.

Modular programming languages enable the creation of modules as units for encapsulation, information hiding and separate compilation. Modules can be interconnected with other modules to create large programs. A language enables modular programming if it provides adequate mechanisms to develop units independently and interconnect them to achieve some functionality. The module system varies in languages. Some of them are more powerful than others, some defines modules as first-class values and some as second-class values. However many module systems of different languages share some commonalities.

Object-oriented languages have honed the programming concepts to just classes and often lack a module system. The benefits of having a module system have been recognized and many researchers are working on the inclusion of a module system in several languages [AZ 01, BAF 03, BPV 98, FF 98, FF 98b, MFH 01, and MFH 02].

In this section we focus primarily on Java, but almost everything applies to C#, or for that matter C++, as well.

Java is a class-based language and it is often considered to be a language with a small number of concepts. It was designed to be simple enough to allow

programmers to learn it easily [GJSB 00]. Java designers included in it ideas from other languages and avoided the inclusion of new or untested features in the language. In the Java programming language, the class is the most important concept. The class supports abstraction, encapsulation, and information hiding. Classes are used to implement many different concepts that are not directly present in Java. Java forces its declarations to belong to the only structuring form available: the class. This leads to overburdening the class mechanism with several incompatible uses.

Not all classes are used to create objects. There are some classes that are not meant to be instantiated, because they are not completely defined e.g., **abstract** classes. There are other classes that are not abstract, they are completely defined, and yet no instances can be generated because their constructors are private, e.g., java.lang.Math. Classes are used in several unrelated ways. This leads to much confusion especially for beginners.

a) **Classes with only a main function.** There are classes that contain only a **main** function. This kind of class is used to contain some declarations and a main function, not to create instances of it, although nothing prevents you from doing so.

b) **Classes act like libraries.** Some classes contain the definition of a set of named constants or related functions. No instantiation of the class is required to execute its methods because these methods are not related

with objects but with the class itself. They are accessed using their fully qualified name. This kind of class acts like a library and requires some annotations to differentiate it from a "normal" class.

c) **Class methods act like procedures**. In object-oriented languages, methods are executed by sending a message to the object that contains the method, but this is not always the case in Java and C#. If a method is declared to be **static**, it has to be executed without sending a message but as a call to a procedure. So there are two different ways to execute a method: sending a message to an object using *object.methodName();* or calling it to execution in the same way you calla procedure using *ClasName.methodName();*.

d) **Not all classes can be extended**. A class can be declared **final** in Java and **sealed** in C#, when its definition is complete and no subclasses are needed. Final/sealed classes cannot be extended by a subclass definition. Final classes never have a subclass.

e) **Not all classes may be used as types**. A class declaration, defines a new reference type of the name of the class. The class type can be used later to declare variables that contain a reference to the class. But not all classes should be used as types. We can declare a variable of type java.lang.Math, but we cannot assign to it. It is a compile-time error trying to apply the **new** operator because Math constructors have private

access. That means we cannot create an instance of the class and it is meaningless to declare a variable of that class type.

5.7.1. Roles of the class in Java.

"*From a class definition, you can create any number of objects that are known as instances of that class*" [AG 98, page 1]. But you do not always create objects from a class definition. The creation of objects is not the only role for classes. Objects are not always needed to solve problems and the class mechanism is overloaded with several roles.

The confusions and inconsistencies listed in the previous section are symptoms of competing roles that classes are asked to play. We try to enumerate these separate roles.

a) **The class as a factoring commonalities mechanism.** Classes are to factor commonalities among many groups.

b) **The class as a specialization mechanism**. Some classes can be extended to specialize behavior or to add new features to the class. A class hierarchy is created when subclasses are defined.

c) **The class as a template for objects**. A class contains different members: fields and methods. A class declaration defines the state and behavior that instances of that class will have. It also defines the constructor method of the class.

d) **The class as a type**. A class declaration declares a class type name. The name of the class is used to define the type of a variable that will contain the object when instantiated. Extending a class generates a subclass, which is a subtype of the class type it extends. The type hierarchy is unified with the class hierarchy.

e) **The class as a compilation unit.** The first program presented in [AG 98] is the class *HelloWorld* which has no members (fields or methods), no object is created form that class definition. The class is used only to contain the special static method ***main***. Many examples that do not require the creation of objects follow this pattern using the class to create a compilation unit with a point to start execution.

f) **The class as container of named constants.** The class is used as an encapsulation mechanism. The elements in this kind of class are defined by declaring variables as **static** and **final** and providing their values in declarations. The class is not used to create objects.

g) **The class as container of functions** (libraries). General functions of the Java runtime system and the underlying operating system are grouped in special classes like Runtime, System, and Math [GJSB 00]. The Math class contains only static constants and methods for common mathematical manipulations. The access to its elements is using the dot

notation with the name of the class followed by the name of the function that wants to be executed.

5.7.2. Modularity problems in object-oriented languages.

Szypersky [S 92] suggested that object-oriented languages should have a modularity mechanism beside the class. The examples used in this section are adapted to Java but they are based on Szypersky's paper [S 92]. These examples show some situations in which the use of traditional modules provides a more appropriate solution to certain kind of problems.

5.7.2.1. Structures that need no local data.

The need to implement structures that need no local data emerges frequently in programming languages. Object-oriented languages that have the class as the only structuring mechanism must provide an extra element or work around to simulate the import mechanism used in other languages. This is the case for C# and Java. The absence of an import mechanism in object-oriented languages affects the readability and clarity of the produced code.

A structure that contains no local data but a set of functions is usually called a library. In this section a library of mathematical functions is defined and we examine three ways to access its elements. A typical definition of a class that contains a library of mathematical functions is shown in figure 5.2. We use a

Java-like syntax, but the class definition excludes the modifiers static and final. This example is intended to show that without these extra elements, classes cannot naturally act like modules.

```
public class Math {
    public final double PI=3.141592;
    public double sin (x: double) { … }
    public double cos (x: double) { … }
    public double tan (x: double) { … }
    …
}
```

Figure 5.2. A typical definition of a class Math.

In classic object-oriented languages a simple way to access the functionality defined in a class is by creating an object of that class and sending a message to it in order to execute a method. Inherited methods are always available and the special variables *this* and *super* can be used to refer to method is the superclass or the object itself. Java provides another way to access a method of a class.

We assume that given the class Math, we need to create a class called Main that needs to access some of the mathematical functions defined in the class Math. We illustrate this with the next three approaches to access a member of the class Math.

a) **Using inheritance**. Figure 5.3, shows this approach. The class Main inherits class Math only to be able to use the functions defined in it. The methods of class Math are accessed using the special variable *this*. Inheritance is primarily a mechanism to specialize classes. Using

inheritance in this situation is misleading because Main is not a specialization of class Math. Inheritance is used only to make available the name space of the library.

```
class Main extends Math {   // refers to class Math in figure 5.2
     …
     public void calculate( ) {
          double x,y;
           y = this.sin(x);
     }
}
```

Figure 5.3. Using inheritance to access a library member.

b) **By Composition**. In this approach, presented in figure 5.4, a variable is defined to contain an instance of the class Math. All the methods of the class Math are available using this variable. We could say that all instances created from class Math are going to be exactly the same because they do not contain state. Therefore instantiating the class is redundant.

```
class Main {
     …
     public void calculate( ) {
          double x,y;
          Math dummy = new Math();  // refers to class Math in figure 5.2
          y = dummy.sin(x);
     }
}
```

Figure 5.4. Using composition to access a library member.

c) **Static class members**. We mentioned in section 5.1.1 that the class in Java could be used as a container of functions. A modifier can precede the definition of a class, a field, and a method. There are several modifiers and they are used to modify the semantics of those elements. The definition of class Math in Java shown in figure 5.5 was taken from [GJSB 00]. This definition of class Math involves the use of the **static** modifier. Java classes classify their members into class members (static) and instance members. All the members of the Math class, which are preceded by the modifier static, are class members. Figure 5.6 shows the implementation of class Main importing[2] the static members of the class java.lang.Math shown in figure 5.5. Class members are invoked without a reference to a particular object or class, but by using the name of the method directly. In this way, a Java program does not need to create an object to execute the methods defined in a class.

```
public final class Math {
   public static final double E=2.7182818284590452354;
   public static final double PI=3.14159265358979323846;
   public static double sin (double a);
   public static double cos (double a);
   public static double tan (double a);
   …
}
```

Figure 5.5. Java definition of class Math in java.lang.Math.

[2] This feature "import static…" will be available in the next major Java release, Tiger v. 1.5. In Java v. 1.4 members of class Math can be invoked using qualified names like Math.sin(x).

```
import static java.lang.Math;     // refers to class Math in figure 5.5

class Main {
    …
   public void calculate () {
        double x,y;
        y = sin(x);
   }
}
```

Figure 5.6. Importing static class members.

These three different ways to access the elements of the library show us that using classes to provide the functionality of modules involves the use of some unnatural elements in the class. We notice that the absence of an import mechanism in object-oriented languages affect the readability and clarity of the code produced.

5.7.2.2. Structures with dependencies on other structures.

The second problem presented by Szyperski in [S 92] is related with preserving invariants. Assume that there are two classes Linkable and LinkedList as listed in figure 5.7.

```
class Linkable {
     Linkable next;                    // non-private access required
     Object node;
     Linkable (Object o) {
          node = o;
          next = null;
     }
}
class LinkedList {
    Linkable head = null;
    void add(Object y) {
          Linkable x = new Linkable(y);
          x.next = head;                    // access next
          head = x;
      }
    boolean empty( ) {    return head == null; }
}
```

Figure 5.7. Separated classes with dependencies.

These two classes are not related by inheritance. Class LinkedList needs to have access to the *next* field of class Linkable. To allow this access, the field *next* must be declared **public**. This violates the encapsulation principle because now any class in the system has access to that field. Classes Linkable and LinkedList are closely related but there is no way to express this relation with flat namespaces. Approaches to solve this problem in C++ and Java are presented next.

a) **Friend functions.** C++ provides friend functions to solve this problem. A friend function is a function defined outside the class and it has access to the elements of the class. The advantage of this approach is that a relation between classes can be established by declaring them to be friends. A disadvantage is that friends have to be declared in

advance. If we need to give access to another function, i.e. declares another friend, the class has to be updated and recompiled.

b) **Packages.** In Java a *package* is a collection of related classes and interfaces providing access protection and namespace management [GJSB 00]. Member variables annotated with *package* access level allows classes in the same package to access the members. This level of access assumes that classes in the same package are trusted friends. Figure 5.8 shows these classes. A disadvantage in this approach is that all classes in the package will have access to member variables. Access cannot be targeted to a specific class or method.

```
package LinkList;

class Linkable {
     Linkable next;                  //  access limited to package
     Object node;
     Linkable (Object o) {
        node = o;
        next = null;
     }
}
public class LinkedList {
    Linkable head = null;
    void add(Object y) {
         Linkable x = new Linkable(y);
         x.next = head;
         head = x;
      }
    boolean empty() {  return head == null;  }
}
```

Figure 5.8. Classes in a package.

c) **Nested classes.** A nested class is a class that is defined inside another class. A syntactic relationship between two classes is established when a class is declared as a nested class of another. Figure 5.9 shows this relationship. Class Linkable is declared as a nested class of LinkedList. This solution is quite serviceable. But the addition of nested classes makes the LinkedList class serve two different roles: incorporation and instantiation.

```
class LinkedList {

   private static class Linkable {        // nested class
       Linkable next;
       Object node;
       Linkable (Object o) {
          node = o;
          next = null;
       }
   }

    Linkable head = null;
    void add(Object y) {
           Linkable x = new Linkable(y);
           x.next = head;
           head = x;
     }
    boolean empty() {
         return head == null;
    }
}
```

Figure 5.9. Inner class.

Modula-3 and Ada 95 provide a module mechanism to solve these problems in a more natural manner, but they do not support object-oriented programming to

the extent that Java and C# does. The presence of records and pointers in those languages overlaps with objects providing more than one way to approach a problem. The resulting mix lacks elegance.

In MOBY [FR 99b] classes and modules are neatly separated. The class mechanism supports only a minimal set of features that are inherently related to classes. And all the features of Java packages are supported by the module mechanism.

Ancona and Zucca criticized the lack of a module system in Java and similar object-oriented languages. They proposed a true module system called JavaMod, which is constructed on the top of a Java-like language [AZ 01]. Their module language provides a construct to define basic modules, which are collections of related classes, module interfaces, which are specification of the services a module provides, and a set of operators like merge, renaming, and hiding, to combine software components. This module language allows expressing generic types, mixin classes, and mutually recursive class definitions defined in independent modules.

The separation of classes and modules is a hard problem because they both share capabilities to abstract and encapsulate information. We aim to separate these two constructs giving each one different properties to provide different features. Our module system should support the following language features:

- Separate the module interface from its implementation.
- Control the visibility of module members outside the module.
- Define the interconnection of module interfaces and implementations.
- Provide support to define and manage namespaces.
- The ability to group definition of types, values, functions, classes, etc.
- Modules shouldn't be first-class values they are not types.

5.8. OOL without generics.

Despite the benefits of having a mechanism to support genericity many object-oriented languages had omitted this feature providing only some of the forms of polymorphism described in section 3.6.

Two of the most popular object-oriented languages, Java and C# do not support parametric polymorphism and its absence is recognized as a defect. Other languages that support parametric polymorphism, like SML, do not support object-oriented features.

Both parametric polymorphism and object-oriented features are important elements that facilitate programming. New programming languages supporting these two elements had been designed recently [BFSG 03, P 01, OW 97]. Several proposals to extend Java and C# were developed recently. The main extension proposed for Java is the inclusion of parametric polymorphism [AFM 97,

BCK+ 01, BCK+ 03, BD 98, BOSW 98a, EKMS 97, V 01b]. ML extensions to include object-oriented features are presented in [FR 99, RV 98].

Users of object-oriented languages without generics have to find a workaround to create generic code. Two distinct approaches can be used to workaround this problem:

- Simulate parametric polymorphism with inclusion polymorphism.
- Write different code for each type.

These two approaches have some disadvantages. When subtype polymorphism is used to simulate parametric polymorphism cast operations must be inserted and they can fail at runtime. Inheritance and genericity are two distinct mechanisms that should be separated in the language. Genericity defines the same code for different types while inheritance defines different code for the same family of types. On the other hand, repeating the same code for different types is not a good alternative due to maintainability costs and memory consumption.

These two alternatives are explained in section 5.8.1 and 5.8.2, which are part of previous work presented in [BS 03].

5.8.1. First approach: using the generic idiom.

A typical way to implement a generic class in Java and C# is using the top element of the class hierarchy of objects that serves as a polymorphic representation. In this way, all the elements derived from that hierarchy might be

manipulated in that class. An example of a stack implemented in Java and C# using this technique is presented in figure 5.10.

Java Object-based Stack

```
public class Stack {
   private Object[] store= new Object[10];
   private int size= 0;
   public void push (Object elem) {
      if (size >= store.length) {
         Object[] tmp = new Object[size*2];
         System.arraycopy (store, 0, tmp, 0,
                           store.length);
         store= tmp;
      }
      store[size++] = elem;
   }
   public Object pop() {
       return store[--size];
   }

   public static void main (String [] args) {
      Stack x = new Stack();
       x.push(new Integer(17));
       Integer y = (Integer) x.pop();
   }
}
```

C# Object-based Stack

```
public class Stack {
   private object[] store= new object[10];
   private int size=0;
   public void Push(object elem) {
      if (size >= store.Length) {
         object[] tmp = new object[size*2];
         Array.Copy(store, tmp, size);
         store = tmp;
      }
      store[size++] = elem;
   }
   public object Pop() {
      return store[--size];
   }

   public static void Main () {
      Stack x = new Stack();
      x.Push(17);
      int y = (int) x.Pop();
   }
}
```

Figure 5.10. A stack in Java and C# using the generic idiom.

The problem with this approach is that the programmer has to keep track of the kind of elements that are stored and to recover them using cast operations. Another problem in Java, but not in C#, is that not all the types are derived from a single topmost class. Hence not all the types can be used in the instantiations, e.g., primitive types, but only those derived from the topmost class.

In figure 5.10 we see that both languages are very similar. In both programs it is necessary to explicitly cast the value that is popped from the stack before it can

be assigned to *y* (last line of code in *main* method of both programs *y = (Integer) x.pop( );* and *y = (int) x.Pop( );*) because the elements stored in the stack are of type *object* and must be cast to *Integer* in Java and to *int* in C# to make them compatible with the type of variable *y* which is receiving the value.

In the Java program the value *17* needs to be explicitly wrapped[3] in the *Integer* type that is derived from the *Object* type before it can be sent as argument to the *push* method. Java separates reference types from primitive types. Primitive types need to be wrapped into reference types in order to be used as arguments in this approach.

In C# the value *17* needs to be dynamically allocated or boxed in order to be used as argument. This boxing, which is an implicit coercion, is automatically inserted by the compiler. C#'s type system is unified in that a value of any type can be treated as an object. Every type in C# is, directly or indirectly, derived from the *object* class, which is the ultimate base class of all types.

The main disadvantage of this approach is that the programmer, in order to recover the elements, must insert cast operations that could fail at runtime.

5.8.2. Second approach: specialized code for each type.

Safer programs without cast operations can be written by specializing the code for each type. A *"copy and paste"* of source code is performed and the types

[3] In the new version of Java (Tiger v.1.5) automatic boxing will be provided and this explicit wrapping won't be necessary because the compiler will make an implicit coercion.

are changed to create a specialized version of the class. Two problems are present in this approach: maintainability costs and 'code bloat'.

Figure 5.11 shows an example of a specialization of stack to work with integer (*int*) types. The difference between this implementation and the previous one shown in figure 5.10 is that the type of the elements has changed from *object* to *int*.

Java Object-based Stack

```
public class Stack {
   private int[] store= new int[10];
   private int size= 0;
   public void push (int elem) {
      if (size >= store.length) {
         int[] tmp = new int[size*2];
         System.arraycopy(store, 0, tmp, 0,
                  store.length);
         store= tmp;
      }
      store[size++] = elem;
   }
   public int pop() {
       return store[--size];
   }

   public static void main (String [] args) {
      Stack x = new Stack();
       x.push(17);
       int y =  x.pop();
   }
}
```

C# Object-based Stack

```
public class Stack {
   private int[] store= new int[10];
   private int size=0;
   public void Push(int elem) {
      if (size >= store.Length) {
         int[] tmp = new int[size*2];
         Array.Copy(store, tmp, size);
         store = tmp;
      }
      store[size++] = elem;
   }
   public int Pop() {
      return store[--size];
   }

   public static void Main () {
      Stack x = new Stack();
      x.Push(17);
      int y =  x.Pop();
   }
}
```

Figure 5.11. Specialization of stack for type int.

Type casting is unnecessary when specialized code is provided. The cast operation on the last line of code is not necessary anymore because the type of the elements that are in the stack are the same as the variable *y,* which receives the

value. It is not necessary to wrap 17 into the *Integer* type because the formal parameter of the method push is a primitive type *int*.

Providing specialized code for each type is not a good option. If a bug is detected the source code of all specialized classes must be updated. Another problem is memory consumption at runtime when many instances of specialized classes are allocated in memory.

Parametric polymorphism is an important feature that increases language expressivity and clarity, and improves program safety. It does not appear currently in either Java or C#, but it will be included soon in both languages [BCK+ 03, C# 02, KS 01]. Although several mechanisms for generics exist, the most likely approach to be used in Java and C# is based on passing types as parameters.

The proposal to add generics to the Java programming language includes two new forms of types: parameterized types and type variables. A homogeneous translation approach by *type erasure* to translate these new elements to Java bytecode is described in [BCK+ 03]. The technique used in the translation erases the type variables and insert cast operations that are guaranteed not to fail at runtime. Due to this technique it is not possible to use primitive types to instantiate parameterized classes. The main advantage of this approach is that only one piece of code exists at execution time for all instances. Some disadvantages are that primitive types cannot be used as type parameters and the code generated is not as efficient as specialized code. An beta version of the new Java compiler supporting

generics is available online[4], the final version is expected to be released in the summer of this year[5]. C# is also in the process of being updated to include generics [C# 02]. C# is translated to an intermediate language (MSIL) that is part of the .NET Common Language Runtime (CLR). The implementation of generics in the .NET platform includes two translation approaches: a homogeneous translation can be shared for all instantiations of reference types and a heterogeneous translation specializes code for those types that are not references [YKS 04, KS 01]. This translation approach provides a balance between efficiency and code explosion. It also makes available the run-time types of parameterized types, which won't be possible in Java.

The mechanisms used to implement parametric polymorphism vary in several programming languages. Modula-3 provides generic units, which can be instantiated to generate a normal module or interface to include in a program. Ada follows a similar approach with generic packages. C++ templates allow creating generic classes that can be instantiated with different types to generate a specialized class that works for that particular type. The heterogeneous translation approach used for templates in C++ has as disadvantage that they can cause code bloat when many instances of the template are needed. Its main advantage is that the code generated is very efficient because it is translated for a specific type. Ada also uses a heterogeneous translation approach to generate instances of generic units.

---

[4] http://.java.sun.com/developer/earlyAccess/adding_generics/
[5] http://java.sun.com/j2se/1.5/index.jsp

Generic programming is a valuable feature that should be supported in any programming language. Our language must have a mechanism to support parametric polymorphism directly. Our goal is to include the necessary elements in the language that support the development of generic programming.

5.9. Language design goals.

In this section we summarize the desired features of our language.

- Simplicity. The language must be simple in order to be easily understood and learned.
- Expressivity. Several language features must be part of the language to support different mechanism.
- Encapsulation. Modules and classes will support various language features without overlapping.
- Modules. Modules and module interfaces will be part of the language. A module construct defines a static closed entity of a group of elements. Module interfaces will describe the interconnection of modules and the elements available for clients.
- Genericity. The concepts needed to support generic programming will be part of the language. They must be typechecked at compile time.
- Classes. A class construct to define the commonalities of a set of objects. Specialization of classes must be done using inheritance.

- Inheritance. A simple inheritance mechanism to support single implementation inheritance and multiple interface inheritance.
- Interfaces. An interface construct to define the visible elements of classes and support structure inheritance.
- Types and Subtypes. Interfaces will specify the type of classes and derived classes will generate subtypes.
- Explicit type annotations. Type annotations will be explicit. They help to make code clearer and document programs.
- Binding. Elements defined in modules will be statically bound. Dynamic binding will be the default for methods in classes.
- Static type checking. To detect type errors at compile time.

Chapter 6.

MOOL.

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

Modules. 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.

Types. There is only one kind of type in MOOL: reference types. Everything is a reference to an object of certain type.

Object-Oriented. MOOL includes common features in object-oriented languages like: classes, inheritance, polymorphism, dynamic dispatch, and late binding.

Objects. Objects are instances of classes that are created dynamically at execution time.

Classes. Classes are templates that encapsulate data and procedures. They have two main roles: extension and instantiation. A class factors commonalities; and can be specialized with inheritance. A class is used to generate objects dynamically. The class mechanism is not used to support namespace management nor visibility control.

Class interfaces. A class interface is used to declare the visible elements of a class.

Polymorphism. Two kinds of universal polymorphism are provided. Parametric polymorphism is supported with parameterized types and type variables. Subtype polymorphism is provided to be able to use objects of a subtype where objects of its supertype are expected.

Subtyping. Nominal subtyping is provided for classes and interfaces, i.e. subclasses generate subtypes.

6.1. Definitions.

The complete definition of the language using an extended BNF grammar and the conventions are presented in the appendix at the end of this document. A brief description of the basic elements of the language is presented in this section. Many of the elements are borrowed from Java [GJSB 00], C# [C# 01], Modula-3 [N 91], and MOBY [FR 99].

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.

Comments. Comments are used to document programs and do not generate code at compile time. Only one kind of comments is supported in MOOL. They are called *single line comments.* A comment starts with the two characters '//', and end with the end-of-line character.

Identifiers. Identifiers are names used to define and refer to some elements in a program such as variables, functions, types, etc. Identifiers must start with a letter, followed by letters, or digits. They can have any length but they cannot be the same as any keyword or reserved word.

Keywords and Reserved words. Keywords and reserved words are words that have a special meaning in the language and cannot be used as identifiers. They are listed in table 6.1.

boolean	false	integer	protected
break	fields	interface	return
case	float	main	shadow
class	for	methods	string
const	function	module	super
constructors	if	new	switch
continue	implements	null	this
default	init	object	true
else	import	of	void
extends	instanceof	override	while

Table 6.1. List of keywords and reserved words.

Variable. A variable is name representing a value of certain type. The type of the variable defines the set of possible values of the variable and the set of operations that can be performed on it. An optional initial value can be specified at declaration or a default value will be assigned. MOOL is strongly typed which means that the set of operations that can be performed on a type is enforced at compile time.

Scope. A module block defines the scope of a program. The scope is the region of the program over which a declaration is valid. Nested scopes can be defined with blocks and redefinitions of identifiers hide the previous definition of them. Identifiers are valid in the scope they are defined.

Static error. A static error is one detected at compile time. These errors are most frequently violations to the language definition (malformed identifiers, bad number or type of arguments in function calls, invocation of a method not supported).

Expression. An expression is a construct in the language in which a combination of operators and operands specify a computation that produces a value.

6.2. Types and Subtypes.

MOOL is a strongly typed language, which means that all expressions are type-consistent and it is guaranteed that the programs accepted by the compiler will execute without type errors [CW 85].

MOOL is a statically typed language, which means that every expression in the language has a statically determined type. Values of different types can be assigned to variables only if their types are compatible. Some rules and the subtype relation between types define type compatibility.

There are some predefined types created to hold numeric and boolean values. There is also a special reference value called **null**. The user can create other reference types. The main reference type in MOOL is the class, but interfaces, arrays, and functions are also reference types. Table 6.2 shows the classification of types in MOOL.

Reference types	Types
Predefined	integer float boolean null
User defined	Class type Interface type Function type Array type

Table 6.2. List of types.

Reference types are allocated on the heap. They hold the address of an element allocated on the heap or **null**.

All types in MOOL belong to a hierarchy of types defined in the language. The top element of the hierarchy of types is called **object**. Object variables are defined with a static type, and a runtime type is assigned at execution time when an object is created. Figure 6.1 shows an example of this.

```
class Point implements IPoint {…}
class ColorPoint extends Point implements ICPoint {…}
Point p;                          // the static type of p is Point
p = new ColorPoint();             // at this time the runtime type of p is ColorPoint
```

Figure 6.1. Static and runtime type in MOOL.

Type *ColorPoint* is a subtype of type *Point* because class *ColorPoint* is a specialization of class *Point* and subclasses define subtypes. See section 6.2.5 for an explanation of the subtyping rules.

6.2.1. Predefined boolean and numeric types.

There are three basic types called value types: **integer**, **float**, and **boolean**. Value types contain a raw value and the space needed to store it depends on its representation. Value types and their range of values are presented in table 6.3.

Type	Range	Comments
integer	-32,767…+32,768	Signed value
float	$\pm 1.5 \times 10^{-45} \ldots \pm 3.4 \times 10^{38}$	
boolean	false…true	Logical value

Table 6.3. Predefined numeric and boolean types in MOOL.

6.2.2. Class interfaces.

Both classes and modules have interfaces. In this section we refer only to interfaces for classes. Module interfaces are defined in section 6.6.1.

A class interface is a type declaration that provides a specification rather than an implementation for its members. 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 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. *ExtendsInterfaces* is an optional part that allows an interface to extend other interfaces. All the interfaces listed in the *ExtendsInterfaces* part are supertypes of the interface being created. The *InterfaceBodyDec* part

declares the members of the interface. Interface members can be constants or methods declarations.

An example of a class interface declaration is shown in figure 6.2.

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

Figure 6.2. A class interface declaration in MOOL.

6.2.3. Classes.

MOOL is a class-based object-oriented language. It 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. Inheritance allows building a hierarchy of classes that can be used as a mechanism for code reuse. 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.

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 6.7.3. *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. A class body declaration is defined as follows.

{ {ClassVariables} [FieldsList] ConstructorsList [MethodsList] **}**

An example of two classes is shown in figure 6.3.

```
class Figure implements IFigure {
  fields
      Point center;
  constructors
     Figure () {center.x = 0; center.y =0;}
     Figure (integer x, y) {
        center.x = x; center.y =y;
     }
  methods
     void move (integer dx, 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 implementation of draw
      }
}
```

Figure 6.3. Two class declarations in MOOL.

6.2.3.1. 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.

6.2.3.2. Fields.

Fields are also called instance variables. Each object has a copy of the fields declared in the class. Fields are initialized explicitly or with default values. 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. Fields and methods members can be accessed inside the class using their names or using the special variable *this* with the dot notation *this.member*.

6.2.3.3. 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.

6.2.3.4. Methods.

Methods are functions defined inside a class. 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 and they are dynamically dispatched.

6.2.3.5. Inheritance.

MOOL provides single implementation inheritance and multiple interface inheritance. A class hierarchy is build with the definition and specialization of classes. By default all classes are derived from a special class called **object**. A class inherits from another class and implements one or more interfaces.

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

6.2.3.6. Class hierarchy.

Classes are organized in a hierarchy with class **object** at the top. The class hierarchy is created defining new classes from existing ones. 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.

6.2.3.7. The special variables *this* and *super*.

The keywords *this* and *super* are special variables to refer to a specific object. These keywords can be used only in the context of instance methods or constructors.

Keyword *super* refers to the immediate superclass. It is used to invoke methods from the superclass.

Keyword *this* refers to the object that received the message in a message invocation or to the one being created in a constructor.

6.2.4. Functions.

A function specifies a group of computations. It receives a set of actual parameters and returns a result. A function type is used to declare variables that hold functions. Functions can be passed as arguments to other functions. A function type and a function declaration are as follows.

function (Type | **void**) Identifier FormalParameters
(Type | **void**) Identifier [TypeParameters] FormalParameters (;|Block)

Functions can be generic. Generic functions are explained in section 6.7.3. A function declaration without a *Block* is used in module interfaces and it specifies the signature of the function. A function declaration with a *Block* is used in module implementations and it contains the signature followed by its implementation. Figure 6.4 shows some examples of functions and functions types.

```
function integer max (integer x, integer y);                    // function type declaration
integer abs (integer n);                                          // in module interface

integer abs (integer n) { if (n < 0) { n = n*(-1);}  return n; }   // in module implementation
integer maxint (integer x,y) { if (x>y) {return x; else return y; } }        // function maxint

max any = maxint;                                                          // function variable
```

Figure 6.4. Functions and function types in MOOL.

6.2.5. Subtyping rules.

In this section we describe the rules to define the subtype relation.

All types in MOOL are derived from **object** including the basic types. This means that all types are subtypes of **object** and **object** is supertype of all types.

anyType <: **object**

We use the symbol <: to denote the subtype relationship between two types. Assume S and T are types; S <: T means that S is a subtype of T and T is a

supertype of S. The subtype relation is reflexive and transitive. These two rules are expressed as follows:

For all types T, T <: T that means any type T is a subtype of itself.
For all types P, Q, R, such that P <: Q and Q <: R implies that P <: R

The subsumption rule states that if an expression has type S then it has also all the types of its supertypes. That means that an expression of a subtype can stand as an element of any of its supertypes. This rule is as follows.

For all types T,S and variable v. If v:S and S<:T then v:T

- **Subtyping for classes and 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.

Definitions of subclasses 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 changes allowed for fields

Covariant – Covariant changes for result type of functions.

Contravariant- Contravariant changes for function arguments.

Suppose we have some declarations of classes and interfaces. Table 6.4 shows the types and subtypes created by each definition.

Declaration	Type	Supertypes	Comments
class A {..}	A	Object A	Inherit from object implicitly
class B **extends** A {...}	B	Object A B	Inherit from A
interface IX {...}	IX	Object IX	
interface IY **extends** IX {...}	IY	Object IX IY	
class C **implements** IY {...}	C	Object IY IX	Inherit from object implicitly
class D **extends** B **implements** IY {...}	D	Object A B IX IY	Inherit from B

Table 6.4. Subtyping relation for classes and interfaces.

- **Subtyping rule for function types.** A function type has the form $\alpha \rightarrow \beta$ where α represents the arguments of the function and β represents the result. The subtype rule for function types is defined as follows.

For all types α,β,α', and β': $\alpha \rightarrow \beta <: \alpha' \rightarrow \beta'$ iff $(\alpha' <: \alpha)$ and $(\beta <: \beta')$.

That means that the function arguments change in a contravariant way while the function result change in a covariant way.

6.3. Expressions.

An expression is a formula to compute a value. Expressions are evaluated by executing the operations in the order established by the precedence of the operators they contain. In this section we present the set of operators of MOOL and how they are combined to create expressions.

6.3.1. Constant expressions.

Constant expressions can be evaluated statically at compile time or at initialization of classes. The values generated by them are constants and they cannot be changed during execution of the program.

6.3.2. Literals.

Literals are representation of primitive values or **null**. Some examples are integer *23*, *-100*, float *3.14*, boolean *true* and *false*, etc. The **null** literal is the only value of the null type.

6.3.3. Operands.

An operand represents a value. Operands are represented in expressions as variables, constants, literals, etc. The type of the operands restricts the operations applied on them.

6.3.4. Function call.

A function call could be an expression or part of it if it returns a value. An example of this is a call to the *max* function, which takes as arguments two integers and return the greatest of them.

6.3.5. Operators.

An operator is a symbol used to define an operation between one or more values. Binary operators are left associative. The set of MOOL operators are classified according to the number of operands involved and the way they are applied.

Primary operators have the highest precedence of all. They refer the selection of values. These operators are classified as member selection and application. They are shown in Table 6.5 and some examples are presented in figure 6.5.

Unary operators are applied to a single operand. They are shown in table 6.6 and some examples are presented in figure 6.6.

Binary operators use two operands as arguments. All binary operators are left associative except the assignment operator. Parenthesis may be used to define explicitly precedence for some operations. Binary operators are presented in table 6.7 and some expressions using binary operators are shown in figure 6.7.

Description	Symbol	Notation
member selection	.	id.member
function	( )	functionName(parameters)
array	[ ]	arrayName[index]
instance creation	new	new ClassName(parameters)

Table 6.5. Primary operators.

```
point.x;                    // field selection
colorPoint.draw();          //method selection
sqr(x);                     // function call
data[i];                    // array element
new Color(16);              // instance creation
```

Figure 6.5. Examples of expressions with primary operators.

Description	Symbols
plus negation	+ -
Bitwise complement	~
Logical complement	!
cast	(Type)
Prefix/postfix increment	++
Prefix/postfix decrement	--

Table 6.6. Unary operators.

```
i++;          // postfix increment
-1            // negation
! done       // logical complement
(Point) cp;  // cast
```

Figure 6.6. Examples of expressions with unary operators.

Description	Symbols
Multiplicative	* / %
Additive	+ -
Relational	> < >= <= instanceof
Equality	== !=
AND	&&
Negation	!
Inclusive OR	||
Assignment	=

Table 6.7. Binary operators.

```
(num <= 0) && (num <= 10)   // and
(i == 1) || (flag)                  // inclusive or
aa = 10;                            // assignement
perimeter = 4 * size    // assignment and multiplicative
```

Figure 6.7. Examples of expressions with binary operators.

6.4. 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. A declaration can be preceded by an access modifier, which makes it available outside the scope of its definition.

6.4.1. Modifiers.

All the elements listed in a module interface are public by default. All elements in a module implementation can be used inside the module.

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.

6.4.2. Constants.

A constant declaration introduces a name for a value. A constant declaration and an example follows.

```
const Type Identifier =  ConstExpression ;
const float PI = 3.14;
```

6.4.3. Variables.

A variable declaration introduces an identifier and its type. An initial value can be defined or a default value will be assigned. Variable declaration and an example follow.

```
Type Identifier [= Expression ] ;
integer x = 0;
```

6.4.4. Functions.

A function declaration defines a function signature in a module interface or a function implementation in a module implementation. A function declaration and an example are shown next.

```
(Type|void) Identifier [TypeParameters] FormalParameters (;|Block)
integer max(integer x, integer y);
```

6.4.5. Types.

Three type declarations are defined: functions, classes, and interfaces. These types are declared as follow.

```
function (Type|void) Identifier FormalParameters;
class Identifier [TypeParameters] [SuperClass] Interfaces ClassBodyDec
class interface Identifier [TypeParameters] [ExtendsInterfaces] InterfaceBodyDec
```

Some examples are presented in figure 6.8.

```
const float PI = 3.14;              // constant declaration
                                     // variables declarations
integer a;
float b = 2.0;
m max = maxInteger;
                                          // functions
function integer m (integer val1, integer val2) ;
integer maxInteger (integer val1, integer val2) {
    if (val1 > val2) { return val1;
    else return val2;
    }
}
class Fraction  implements IFraction {      // class declaration
   fields
     integer num = 0;
     integer den = 1;
   constructors
     Fraction(integer n, d) {num = n;  den = d; }
   methods
     …
}
```

Figure 6.8. Examples of constant, variables, functions, and types.

6.4.6. The *import* declaration.

An import declaration makes available all the elements listed in the module interface to be used in the module. Some examples of import statements are presented in figure 6.9. An import declaration has the form:

import Identifier [.Identifier] [as Identifier];

```
import System;
import Math;
import Stack;
```

Figure 6.9. Examples of the import declaration.

6.5. 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. In this section we present the statements supported in MOOL.

6.5.1. Assignment statement.

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. The assignment is valid if the type of the receiver can hold the type of the value generated. The general form of an assignment is as follows:

Expression **=** Expression

If the types of the LHS and RHS expressions are not compatible, an error is signaled. Some examples of assignments are presented in figure 6.10.

```
days[1] = “Monday”
price = 3.99
point.x = 2
i++                    // the meaning of this is  i = i + 1
```

Figure 6.10. Examples of the assignment statement.

6.5.2. Function call statement.

A function call has the form:

Expression (ActualParameters)

ActualParameters is a sequence of zero or more values, which are going to be assigned to the formal parameters defined in the function using positional binding. The parameters are passed by value. In the case of **integer**, **float**, and **boolean** types, the values are copied to the formal parameters such that changes to them do not affect the actual arguments. Other reference types are also passed by value, but changes to the formal parameters will be reflected in the actual parameters because the formal parameters become aliases.

6.5.3. Sequential composition.

Statements are executed sequentially in the order they are defined unless an error occurs. A sequence of statements separates each statement with a semicolon. Figure 6.11 shows a sequence of statements.

```
c= a+b;
x = square(a);
printLine(o.Length);
```

Figure 6.11. Sequence of statements.

6.5.4. Block statement.

A block statement is delimited by curly brackets and may contain local variable declarations and a sequence of statements. A block is treated as one single statement and it is executed by executing in order each statement or declaration contained in it. A block is defined as follows:

Block
{ {LocalVariableDeclaration} Statements **}**

The block statement defines a scope where the local variables declared inside are valid.

6.5.5. Selection.

There are two selection statements in MOOL. The first one is the traditional if statement which allows the selection of one of two possible statements. The other one is a switch statement, which allows the selection of one of several statements. These two selection statements are presented in the next two sections.

6.5.5.1. The *if* statement.

The **if** statement contains an expression and a body, delimited by curly brackets. The body contains a statement and an optional else part. The execution of the if statement is as follows: the expression is evaluated and yields a boolean result, if the result is true then the first statement inside the body is executed, if the

result is false then the statement after the else is executed. In both cases, the execution continues with the statement after the body of the if statement.

An *if* statement is defined as follows.

if (Expression) **{** Statement **;** [**else** Statement] **}**

An example of the *if* statement is shown in figure 6.12.

6.5.5.2. The *switch* statement.

A switch statement contains an expression and a body. The body defines a set of cases and a default clause. The set of cases defines the values for which specific actions are defined. In a switch statement the expression is evaluated and then depending on its value a matching case is selected to continue the execution of the program. If no matching case exists, then the default clause is selected. In both cases the execution continues with the next statement following the switch statement. The value generated by the expression must be integer or boolean.

The switch statement definition is as follows:

```
  switch (Expression) SwitchBlock
SwitchBlock
  { {Case} DefaultStatement }
Case
  case ConstList : Statement
DefaultStatement
  default : Statement
```

An example of a switch statement is presented in figure 6.12.

```
if ( a > b ) { c = a;
else  c = b;
};

switch (args) {
   case  0 : p = new X( )
   case  1 : p = new Y(args)
   default : p = new Object( )
}
```

Figure 6.12. Examples of if and switch statements.

6.5.6. Repetition.

Two repetition statements are part of MOOL. They are called *for* and *while*.

6.5.6.1. The *for* statement.

The for statement contains a controlling-loop part and a block.The controlling-loop part is delimited by parenthesis and contain three parts (ForInit, Expression, and ForUpdate) separated by semicolons. The execution of the for statement is a s follow. First it executes the initialization part, and then evaluates the expression. If the expression yields a true value, the block is executed. When the block is finished the for update part is executed and the expression is evaluated again. The block is executed repeating the process until the values of the expression is false. Then the execution continues in next statement following the block. An example is shown in figure 6.13. The *for* statements has the form:

for ([ForInit]; [Expression] ; [ForUpdate]) Block

6.5.6.2. The *while* statement.

The while statement executes the expression and if its value is true executes the block that is part of it coming back to evaluate the expression and repeat the process until the expression gets a false value. The while statement has the form:

while (Expression) Block

An example of a while statement is presented in figure 6.13.

```
for (integer i=0; i < max; i++) {
    IO.printLine(i) }

while (i < max ) {
   IO.printLine(i);
   i++;
};
```

Figure 6.13. Examples of for and while statements.

6.5.7. The *continue*, *return*, and *break* statements.

The *break* statement ends a loop execution and continue the execution in the next statement. The *continue* statement returns the execution to the control loop part. The *return* statement is used to finalize the execution of a function, defining the value to be returned.

These statements are defined as follows.

```
continue
return Expression;
break
```

6.6. Modules and module interfaces.

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. Modules are units to encapsulate elements, hide information and separate compilation.

Modules contain two parts: a *module interface* describes the signature of the module and the *module implementation* contains the implementation of the signature.

Modules define the namespace structure to refer to qualified names. They define two scopes: internal and external. Module interfaces show the elements of the module that are visible outside the module. By default all members of a module interface are public. Some members of classes can be annotated as protected, in that case members are available only for specialization. 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 an initialization part (**init** Block), which is used to initialize the elements of the module before they are loaded to execution.

6.6.1. 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.

A module interface declaration has the form:

module interface Identifier ModuleBlock

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. This module interface is presented in figure 6.14.

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

Figure 6.14. Example of a module interface.

6.6.2. Module implementation.

A module can implement several module interfaces. The interfaces implemented by a module are exported by it. A module implementation can contain constants, variables, and types (functions, classes, and interfaces).

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

An example of a module implementation is presented in figure 6.15.

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

Figure 6.15. A module implementation.

6.6.3. Classes inside modules.

Classes have three main purposes; factor commonalities, serve as a base for specialization, and create objects. On the other hand the main role of modules is to be used as organizational units and to create scopes.

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.

We illustrate this in the example shown in figure 6.16.

```
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
}
```

Figure 6.16. A module interface and a module implementation.

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 in figure 6.16 contains a definition of class *C1* with three methods: *m1*, *m2*, and *m3*. Tables 6.8 and 6.9 show how these members are

available for users and derived classes inside and outside the module implementation.

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

Table 6.8. Visibility of methods inside the module implementation.

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

Table 6.9. Visibility of methods outside the module implementation.

6.6.4. Separate compilation.

Module interfaces make separate compilation type-safe. They help to keep large programs well structured and they provide a mean to hide implementation and avoid dependencies.

6.7. Generics.

Generics are abstractions over types. MOOL provides support for the definition of generic types, and type variables. Generic types are also called

parameterized types. They could be parameterized classes, parameterized interfaces, and parameterized functions.

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.

6.7.1. 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. Types are sent as parameters to create an instance of the parameterized type.

Type variables are defined after the identifier of the type declaration and they can contain bounds to other types to constraint the type that can be used in instantiations. Section 6.7.2 describes different kinds of constraints.

Figure 6.17 shows an example of two parameterized types that contain a type variable. A type variable called *T* is used in the declaration of the parameterized types *IList* and *List*. The type parameter *T* is enclosed in < > and it defines that the class is a parameterized type that receives a type parameter to create instances of it.

```
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) {…  }
  …
}
```

Figure 6.17. Examples of unconstrained type variable.

6.7.2. Type constraints.

Generic code can be defined for all the types available in the system or for some of them that hold some properties. The former is called *unconstrained genericity* and the later is called *constrained genericity*. [M 86]

6.7.2.1. Unconstrained genericity.

Generic code that can be instantiated with any type available in the system is called unconstrained. An example of using unconstrained genericity is the generic class presented in figure 6.17. The type variable *T* used in there does not contain constraints. That means that the generic class *List* can be instantiated using as actual type parameter any type available in the system. Some examples of instances of *List<T>* are shown in figure 6.18.

```
List< string > names = new List < string >( );
List< Point > pointList = new List< Point >( );
List<integer> values = new List<integer>();
```

Figure 6.18. Some instantiations of class List<T>.

6.7.2.2. Constrained genericity.

When the types used to instantiate generic types need to be bound to other types it is called constrained genericity.

In the example shown in figure 6.19 the type variable is bound to an interface and as result the types used in the instantiation are constrained to hold this relationship. In this example, class *Point* implements the interface *IOrderable*. Class *Point* can be used as a type parameter to create an instance of the generic class *OrderedList* because class *Point* is bounded to *IOrderable* and that is a requisite of the type parameter of class *OrderedList*.

```
interface IOrderable {
    integer compareTo(object elem);
}
interface IOrderedList < T > {
    T remove();
    void insert (T elem);
}
class OrderedList< T implements IOrderable > implements IOrderedList <T > {...}
class Point implements IOrderable { .... }
OrderedList<Point> olp = new OrderedList<Point>();
```

Figure 6.19. A parameterized class with a constrained type parameter.

It is possible to restrict the type parameters using recursive bounds. This kind of bound is useful when a binary method is defined inside the interface or class and we want to restrict the type of the actual parameters of that method to be the same type as the object that receives the message.

We change some elements of figure 6.19 to define a class with a recursive bound, which is shown in figure 6.20.

```
interface IOrderable <T> {
    integer compareTo(T elem);
}
class OrderedList< T implements IOrderable<T> > implements IOrderedList <T > {...}

class Point implements IOrderable <Point> { .... }

OrderedList<Point> olp = new OrderedList<Point>();
```

Figure 6.20. A class with a recursively bound type parameter.

6.7.3. 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. A generic function is called in a similar way to that of a non-generic function, except for the type parameters. In the example shown in figure 6.21 the

functions *swap* is a generic function. Function *swap* receives a type parameter called *T* and three formal parameters

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

Figure 6.21. An example of a parameterized function.

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

class Identifier [TypeParameters] [SuperClass] Interfaces ClassBodyDec

Examples of the definition of a generic class are shown in figures 6.17, 6.19, and 6.20.

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

class interface Identifier [TypeParameters] [ExtendsInterfaces] InterfaceBodyDec

Examples of generic class interfaces are *IOrderedList* of figure 6.19 and *IOrderable* of figure 6.20.

6.7.4. Subtyping rules for parameterized types.

The direct supertypes of a parameterized class are:

- The type listed in the extends clause and
- The types listed in the implements clause.

The direct supertypes of a type variable are the types listed in its bounds.

The subtypes of a type T are those that have type T as a supertype.

Subtyping does not extend through parameterized types. This means that if *S* and *T* are types and *C* is a parameterized type, if *S<:T* does not imply *C<S>* <: *C<T>*.

Chapter 7.

Language Evaluation.

"Language design is decision making."
Niklaus Wirth.

Language design is a complex activity filled with tradeoffs. Language features that have proved their success in some languages are not always suitable for other languages.

Wirth gave us a list of demands a language designer frequently encounters when designing a programming language [W 87b]. Some of them are closely related with the language itself while some others are related with the translator (compiler) used to generate executable code. However, as he noticed, some of those points are contradictory and it is the designer's responsibility to decide where to put the emphasis. On the other hand, Hoare argues that good language design can be summarized in five phrases: simplicity, security, fast translation, efficient object code, and readability [H 87].

The characteristics most crucial in the design of a programming language are simplicity and safety.

The design of MOOL was driven with these two features in mind. Simplicity as the main feature of the language will allow programmers to easily learn and use the language, while safety provides a guarantee that errors are going to be detected at either compile time or runtime.

Some languages equate simplicity with the number of different concepts they provide. However reducing the number of concepts to a minimum may require the same construct being used for several purposes, which leads to an unnatural way to express different abstractions. We aim to separate classes and modules, which are two concepts that have been thrown together in modern object-oriented languages. We believe that these two constructs will allow us to express two different abstractions in a more natural way.

On the other hand, there are languages that provide different constructs for every different kind of abstraction burdening the language with many concepts that are difficult to understand and which can be used to approach several solutions for the same problem. Users of these kind of languages, tend to master only a subset of the language. We limited the inclusion of many constructs in order to maintain a balance of concepts with well-defined roles.

The flexibility provided by some languages has a price, which is usually the lack of safety at runtime. Unsafe languages do not guarantee that programs accepted by the compiler are free from runtime errors while safe languages are those that protect their abstractions guaranteeing that programs accepted by the

compiler are going to execute free of type errors. Types play an important role in language safety. They can be statically or dynamically checked to achieve language safety, although some runtime checks are performed in statically typechecked languages, e.g., array-bounds or downcast operations.

In previous chapters we pointed out problems in other languages. Now we compare how well MOOL does in solving these problems.

7.1. Methodology.

In this section we describe the process we are going to follow to compare the features of MOOL with respect to other programming languages with similar characteristics. We aim to evaluate MOOL using an ad hoc methodology.

There can be no qualitative measure since the complexity or simplicity of a construct is relative to some extent. Instead we have written example programs illustrating that the combination of features in MOOL are more natural overall that existing languages.

Our language, presented in chapter 6, contains a combination of features of modular and object-oriented languages, and allows the definition of generic code.

It is not our intention to describe or evaluate every feature or construct in the language since some of them are borrowed from other languages and are well understood. Our evaluation process is conducted only for some features that

distinguish MOOL from other programming languages. This process has three basic steps:

1. Identify some language features that are needed to support modular, object-oriented or generic programming.
2. Describe how these features are supported in MOOL using a program.
3. Compare the solution in MOOL with respect to other languages and list the advantages and disadvantages that MOOL's solution offers.

7.2. Examples approached in MOOL.

We selected six issues grouped into three areas. They are listed as follows:

- Modularity .
    - Structures that need no local data. (Section 5.7.2.1)
    - Structures with dependencies on other structures (Section 5.7.2.2)
- Genericity.
    - Generic classes and instances. (Section 5.8)
    - A generic sort function.
- Object-Oriented.
    - Inheritance and binary methods. (Section 3.7)
    - Implementing mixin inheritance. (Section 3.5.3)

We approach each of these problems in the next sections.

7.2.1. Structures that need no local data.

A library is defined with modules in MOOL. A module interface contains the definition of all the elements that are available for clients. A module implementation provides code for all elements listed in the interface it implements. The elements of a module are statically allocated. We restrict visibility and hide implementation by providing the module interface and implementation in separated files.

Figure 7.1 shows the module interface and module implementation of a library of mathematical functions.

```
// Set of mathematical functions
//  and numeric constants

module interface IMath {

    const integer MAXINT = 32565;
    const float E = 2.71828;
    const float PI = 3.14159;

    integer abs (integer x) ;
    float sin (float x) ;
    float cos (float x) ;
  …
}
```

```
// Implemenattion of interface IMath

module Math implements IMath {
   //implements absolute value function
    integer abs (integer x) {
      if (x<0) { x=  x*(-1); }
      return x;
    }
  // implements sin function
    float sin (float x) {…}
   // implements cos function
    float cos (float x) {… }
  …
}
```

Figure 7.1. Library of mathematical functions in MOOL.

All the elements listed in the module interface are available for clients of the module. All members of the module implementation that are listed in the module interface keep the same access modifier defined in the interface. Constant

elements are defined only once either in the module interface or module implementation. New elements can be introduced in the implementation but they will not be available for users who import the module because they do not appear in the module interface.

A program that uses the library of mathematical functions is shown in figure 7.2.

```
module Test implements IMain {
    import Math;
    void main (string [] args) {
        float x = 3.5;
        float y;
        …
        y = sin(x);
    }
}
```

Figure 7.2. A program using the library of mathematical functions.

The module *Test* imports the library *Math.* All the functions and constants defined in the module interface *IMath* are available in this program. No extra mechanisms are needed to access the elements of the Math library.

The Java version for this library is shown in the left part of figure 7.3. In the right part, there are two versions of a program that uses the library.

```
public final class Math {
  public static final double
                E=2.7182818284590452354;
  public static final double
                PI=3.14159265358979323846;
  public static double sin (double a);
  public static double cos (double a);
  public static double tan (double a);
  …
}
```

```
import java.lang.Math;
class Test {
   public void calculate () {
        double x,y;
        y = Math.sin(x); // using qualified name
   }
}
-------------------------------------------------
// using import static
import static java.lang.Math;
class Test {
     public void calculate () {
        double x,y;
        y = sin(x);
   }
}
```

Figure 7.3. Java's Math library and two programs using the library.

The Java library *Math* contains some special annotations that modify the semantic of the class definition. The class *Math* is annotated as **final**, which means it cannot be extended. No instances of this class can be created because no constructors are provided. Members of the class are annotated as **public static final** to define constants, and as **public static** to define methods that act like procedures.

In the right part of figure 7.3 there are two versions of a program that uses the library. The one at the top import the elements of the library but requires specifying each function with its fully qualified name. The one at the bottom uses a new import mechanism that will be available in the next version of Java (1.5) [BG 03, JSR 201]. The **import static** mechanism allows importing static members of a class and referring to them without their fully qualified name.

The main advantage of Java in this respect is that Java has only one structuring mechanism (the class) which is used to simulate the behavior of modules. Its main disadvantage is that all the annotations required in the class and its members obfuscate its meaning. None of the special annotations used in Java for the class, its members or the import declaration are required in the MOOL being this an advantage of MOOL over Java and other object-oriented languages that have only classes. The semantic definition of modules declares them as containers that encapsulate other elements and hide information. Visibility is controlled by the module interface and the import declaration makes the elements of the imported module available to use.

7.2.2. Structures with dependencies on other structures.

Sometimes the implementation of a structure requires the use of other structures. These two structures can be defined in the same module providing access to their elements without violating encapsulation.

In this section we present a MOOL implementation of a linked list. We use the same example as the one presented in section 5.7.2.2.

In the left of figure 7.4 is the module interface that describes the class *LinkedList*, which can be used to create linked lists of any kind of object. The module implementation is in the right of figure 7.4.

The implementation contains an extra class named *Linkable*, with two fields, a constructor, and no methods. This class is used to define the nodes of the linked list in a structure containing the element and the link to the next element.

The class *LinkedList* has access to the fields of class *Linkable* because they are defined in the same module. But class *Linkable* is not part of the module interface so its scope is limited to the module implementation.

Our module construct is a container that encapsulates elements and hide some of them by providing a module interface that list only those elements that are available for users. No implementation details are revealed.

```
module interface ILinkList {
     class interface ILinkedList {
        object remove ();
        void add(object elem);
        boolean empty ();
     }
    class LinkedList implements ILinkedList{
       constructors
          LinkedList ( );
    }
}
```

```
module LinkList; implements ILinkList
    class interface ILinkable { }
    class Linkable implements ILinkable {
     fields
        object node = null;
        Linkable next = null;
     constructors
         Linkable (object elem) { node = elem; }
    }  // class Linkable

    class LinkedList {
     fields
        Linkable head= null;
     constructors
        LinkedList ( ){ }
     methods
        object remove() {
           object temp = head;
            if (head == null) { return null;
            else {  head = head.next;
                  return temp;  }
            }
        }
        void add(object elem) {
           Linkable n = new Linkable(elem);
           n.next = head;
           this.head = n;
         }
        boolean empty( ) {
            return (this.head == null);
         }
    }  // class LinkedList
}
```

Figure 7.4. A module interface and implementation of a linked list.

Java provides two different solutions for this problem, which were presented in section 5.7.2.2. The first one defines package scope for the elements of classes that are in the same package. The access to the elements of a class is not restricted to a specific class but to all classes that are in the package. The second

solution (nested classes) allows using the class as a container of other classes modifying its semantic.

Consistency in the use of modules is the main advantage of MOOL. Modules have a well-defined role and are used in a consistent way without need to provide extra annotations or elements that change their semantics.

Despite the similarities of modules and classes as units of encapsulation and information hiding, it is possible to separate them using two distinct constructs with specific roles. We believe that it is better two have two different constructs than to have only one with many modifiers. This separation causes less confusion.

7.2.3. Generic classes and interfaces.

In this section we define a generic stack using unconstrained genericity. The example contains: a module interface to describe the stack signature, a module implementation to define the generic stack, and a client program which uses the stack to create instances of it.

Figure 7.5 contains the module interface and implementation of a generic stack. The module interface describes a generic class interface *IGenStack* and a generic class *GenStack*. Class interface *IGenStack* and class *GenStack* have both a type parameter *T* with no bounds, which means that any type can be used as argument to create an instance of the class. The type variable *T* is used to define the type of some elements of the class and the interface.

```
module interface MIGenStack {
   class interface IGenStack <T>{
      T  pop();
      void push(T elem);
      boolean empty ();
   }
  class GenStack<T> implements IGenStack<T> {
    constructors
      GenStack<T> ();
      GenStack<T>( integer max);
  }
}
```

```
module MGenStack implements MIGenStack {
  import Array;
  class GenStack<T> {
    fields
      T[] store = new T[10];
      integer size = 0;
    constructors
      GenStack<T>( ){ return this; }
      GenStack<T>(integer max){
                store = new T[max];
                return this; }
    methods
       T pop() { return store[--size]; }
       void push(T elem) {
          if (size >= store.length()) {
              { T [] tmp = new T[size*2];
                copy(store, tmp, size);
                store = tmp;
              }
          }
          store[size++] = elem;
       }
       boolean empty() {
          return (size == 0);
       }
  }  // class GenStack
} // module
```

Figure 7.5. A module interface and implementation of a generic stack.

In the module implementation the generic class *GenStack* contains two constructors to define instances of the class providing any type argument. These two constructors have different signatures. The first one does not have arguments and the class fields are going to retain their initial values. The second one receives an integer, which is used to define the initial size of the stack. The class contains also the implementation of all the methods defined in the interface using the type variable T to define the type of some elements.

An example of a program using the generic class *GenStack* is shown in figure 7.6.

```
module TestStack implements IMain {
  import MGenStack;
  void main () {
    GenStack <integer> iStack = new GenStack<integer>();  // creates a stack of 10 integer elements
    GenStack<string> sStack = new GenStack< string >(100);    // creates a stack of 100 float elements

    iStack.push(17);
    integer y = iStack.pop();    // cast operation is not needed before assignment
    …
    sStack.push("hello"); …
  }
}.
```

Figure 7.6. Creating instances of a generic stack class.

In this program, the generic class *GenStack* is instantiated twice. The first instance of stack creates a stack of integers with space to hold initially 10 elements. The second instance creates a stack of string elements that have initially space for 100 elements.

Neither Java nor C# support the definition of generic types as they are now. Both languages have plans to release new versions of the languages that include generics types [BG 03, C# 02]. In section 5.8 we described two approaches followed in Java and C# to implement generic code.

The first approach uses the generic idiom to simulate parametric polymorphism with subtype polymorphism but this approach requires the use of cast operations to recover the elements from the stack and there is no warranty that all the elements of a stack are of the same type, since any kind of element can be

inserted. In addition Java do not support the use of primitive types as elements of the stack, because they are not unified with the type Object.

The second approach provides specialized code for every type, which is inappropriate for maintenance. Besides maintainability cost, the approach generates many copies of the same code at runtime. The advantages of this approach are that no cast operations are needed and the specialized code has good performance at runtime.

The design of generics for Java has some constraints [JSR 014] that affect the final result, i.e. exact types of generic types are not available at runtime, primitive types cannot be used as type parameters, etc. A summary of the features of the genericity mechanism is presented in [BS 03].

The genericity mechanism of MOOL is based on that of Java. However MOOL does not suffer the restrictions imposed in Java because we do not have to preserve compatibility. All types can be used as type parameters to instantiate generic classes because all types belong to the same hierarchy of types.

We haven't described the translation approach but it seems feasible to implement a hybrid translation similar to the one for C# presented in [YKS 04, KS 01].

7.2.4. A generic sort function.

A generic sort function of arrays of any type is described in this section.

Figure 7.7 shows a module interface that contains the declaration of a generic function signature. The generic function *sort* has a type parameter *T* recursively bounded to an interface named *IComparable*. Function *sort* receives as argument an array of elements of type *T*.

```
// class interface IComparable is defined in
// module GenInterfaces
//    class interface IComparable <T> {
//          integer compare (T elem);
//    }

module interface IGenericSort {
   import GenInterfaces;
   void sort <T implements
                  IComparable<T>>(T[] d);
}
```

```
module GenSort implements IGenericSort {
   void sort <T implements IComparable<T>> (T[] d){
      boolean done = false;
      while (!done) {
         done = true;
         for (integer i=0; i<data.length() -1; i++) {
            if ( data[i].compare( data[i+1]) < 0) {
               { T temp = new T (data[i]);
                  data[i] = data[i+1];
                  data[i+1] = temp;
                  done = false;
               }
            }
         } //for
      } //while
   } // sort
} // module
```

Figure 7.7. A module interface and implementation of a generic sort.

The module implementation of the generic function *sort* is in the right of figure 7.7. Any type implementing the interface *IComparable* for itself can be used as type parameter to instantiate the generic function *sort*.

An example of a program using the generic function *sort* is presented in figure 7.8. The *main* function of module *SortingData* defines an array of integer elements named *scores*. After the initialization process, the generic function *sort* is instantiated with a type parameter *integer*, which is the type of the elements of

scores. The function is executed with the array *scores* as actual parameter. Finally the elements of the array are printed.

```
module SortingData implements IMain {
  import System, GenSort;

 // we assume that integer implements IComparable<integer>

  void main( ) {
    integer [ ] scores = new integer [100];
   //   put data in array scores
    for (integer i=0; i<scores.length() -1; i++) {
     …IO.read(scores[i]);
    }
    sort<integer> (scores);
    // print the elements of array scores
    for (integer i=0; i<scores.length() -1; i++) {
     …IO.printLine(" Position "+ i + " Value " + scores[i]);
    }
  }
}
```

Figure 7.8. A module implementation using a generic function.

Java allows the definition of generic methods inside classes, which may not be parameterized. Generic methods can be annotated as static to act like functions. The instantiation of a generic method requires no type parameters because they are inferred from the arguments. For every call of the generic method the compiler will "*infer the most specific type argument that make the call type-correct*" [B 04].

In MOOL generic functions are elements described in modules. No special annotations are needed for the functions or for the call of them. Generic functions can be instantiated in any module that imports the module where the generic function is implemented.

7.2.5. Inheritance and binary methods.

The type system of object-oriented languages with nominal subtyping poses a problem when classes with binary methods are used to create subclasses. Section 2.8 shows the problems described by Cook et al. in [CHC 90].

MOOL is an object-oriented language with nominal subtyping and single dispatch. In order to preserve type safety, we adhere to the rules described to allow changes of types in subclasses. If a subclass overrides a method, the types of the parameters can change only in contravariant way.

This decision restrict the expressiveness of the language because the arguments of methods can not change covariantly but preserves type safety as in the case of Java and C#. A disadvantage of this restriction is that binary methods are difficult to implement in subclasses and it is the responsibility of the programmer to implement them correctly.

An example of a class with a binary method and a subclass is presented in figure 7.9.

The argument of the *equal* method cannot be changed in the subclass and if we want to compare objects of these two classes, *Point* and *ColorPoint*, some extra operations are needed as shown in the *equal* method of class *ColorPoint*.

```
module interface IBinaryMethods {
  import MColor;
   class interface IPoint {
      integer getX();
      integer getY();
      boolean equal (IPoint p);
   }
   class interface IColorPoint extends IPoint {
      Color getColor();
  }
  class Point implements IPoint {
     constructors
       Point ();
       Point ( integer x, integer y);
  }
  class ColorPoint extends Point implements
IColorPoint {
     constructors
       ColorPoint ();
       ColorPoint ( integer x, integer y, Color c);
  }
}
```

```
module MBinaryMethods implements IBinaryMethods {
  import MColor;
   class Point {
     fields
       integer x=0;
       integer y=0;
     constructors
       Point () { return this;}
       Point ( integer x, integer y) {
                this.x = x; this.y = y; return this;}
     methods
      integer getX() { return this.x; }
      integer getY();{ return this.y; }
      boolean equal (IPoint p) {
               return (this.x==x && this.y==y); }
  }
  class ColorPoint {
     fields
       Color c=Color.BLACK;
     constructors
       ColorPoint (); {super; return this; }
       ColorPoint ( integer x, integer y, Color c) {
           super(x,y);  this.c=c; return this;  }
     methods
      Color getColor() { return this.c;}
      overrides boolean equal(IPoint p){
           if (p instanceof ColorPoint ) {return
               (super.equal(p)&&this.c==(ColorPoint)p.c);
           else   return super.equal(p);
           }
       }
  }

  void main (String[] args) {
     Point p = new Point(1,1);
     ColorPoint cp = new ColorPoint(2,2,Color.RED);
     p.equal(cp);
    cp.equal(p);
   …
  }
}
```

Figure 7.9. Inheritance and binary methods in MOOL.

7.2.6. A problem with mixin inheritance.

The example of mixin inheritance presented in section 2.6.3 cannot be directly represented in MOOL due to limitations of our inheritance mechanism. A mixin is a special kind of "class" that is partially defined. It can be mixed with regular classes to generate new classes but it cannot be used for instantiation.

Mixins could be included in MOOL by creating a new construct to define them and a mechanism to restrict the kind of classes that can be mixed to create new classes.

MOOL contains a simple inheritance mechanism, which allows simple implementation inheritance (a subclass has at most one parent class) and multiple interface inheritance (a class can implement several interfaces). We use these two inheritance properties to implement the example of section 2.6.3 in MOOL. Figure 7.10 shows this implementation.

The class interface *IUndo* extends the class interface *IText* inheriting its signature. The class *Textbox* is defined to implement class interface *IText*. A new class *TextboxWithUndo* is defined by inheriting the implementation of class *Textbox* and the interface *IUndo*. There is no conflict in the inheritance of these two elements because both *Textbox* and *IUndo* descend from *IText*.

The resulting class *TextboxWithUndo* has the same behavior in both approaches (MOOL and Jam). The advantage of Jam is that the mixin *Undo* can be

mixed many times with different classes reusing the same implementation, which is not possible in MOOL because we cannot inherit from multiple classes.

```
class interface IText {
  string getText( );
  void setText(string s);
}
class interface IUndo extends IText {
  void undo();
}

class Textbox implements IText {
  fields
    string text;
  constructors
    Textbox() { this.text = null; }
    Textbox(string s) { this.text = s; }
  methods
    string getText( ) { return this.text; }
    void setText(string s) { this.text = s;}
}

class TextboxWithUndo extends Textbox implements IUndo {
  fields
    string lasttext;
  constructors
    TextboxWithUndo() { super; this.lasttext = null; }
    TextboxWithUndo(string s) { super(s); this.lasttext = null; }
  methods
    void undo() { this.setText(lasttext); }
    overrides void setText(string s) {  lasttext = getText( ) ;super.setText(s); }
 }
```

Figure 7.10. Implementation of a mixin class in MOOL.

Mixins represent another kind of abstraction that can be used in programming languages. Many studies related with mixin-based programming had been conducted [BC 90, BL 91, B 92, FKF 98,.ALZ 03] but no production language has implemented them.

Chapter 8.

Conclusions.

We have designed 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 classes, class interfaces and functions.

Our language is similar to other programming languages in many ways. We adopted a similar 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.

8.1. The traditional "*HelloWorld*" program.

The traditional "hello world" program is presented in figure 8.1. There are two implementations, one in MOOL and another one in Java taken from [AG 98].

```
module HelloWorld implements IMain {
    import System;
    void main (string [] args) {
      IO.printLine("Hello World!");
  }
}
```

```
class HelloWorld {
    public static void main (String [] args) {
      System.out.println("Hello World!");
  }
}
```

Figure 8.1. Comparing "Hello world" in MOOL and Java.

In MOOL the *HelloWorld* program is defined using a module that implements the predefined module interface *IMain*, which contains only the definition of the *main* function. The *HelloWorld* module imports the library *System*, which contains a set of input/output operations. The program starts its execution with the first line of the body of the *main* function. A call to the function *printLine* with a string literal as actual argument is executed.

In Java the *HelloWorld* program is implemented in a class that contains only a *main* function, which is annotated with two modifiers: **public** and **static**. The class is not meant to be a template to generate objects; it doesn't contain fields or methods. But there is no other way to implement this program in Java because the class is the only structuring mechanism available.

The Java version of the *HelloWorld* program had provoked a debate among Computer Science professors in several universities [W 01, W 02, XB 03]. Is the *HelloWorld* program adequate to start teaching object-oriented programming? Should the program be changed to define a class with a method and then create another program (class) that creates an object and send a message to execution?. Left part of figure 8.2 shows these two programs taken from [W 01].

The method *printHello* of class *HelloWorld* in left of figure 7.2 is annotated as **public static,** which makes it a class member, and it can be executed by sending a message to an object of class *HelloWorld* (see right part of figure 8.2) or directly using a fully qualified name *HelloWorld.printHello();* as noted in [W 02].

```
class HelloWorld {
   public static void printHello () {
      System.out.println("Hello, World!");
  }
}
class UseHello {
   public static void main (String [] args){
      HelloWorld myHello = new HelloWorld();
      myHello.printHello();
  }
}
```

```
class UseHello {
   public static void main (String [] args) {
      HelloWorld.myHello;
  }
}
```

Figure 8.2. New version of "Hello world" program.

Is the new program (*UseHello*) object-oriented? It seems that the program *UseHello* suffer the exact same problem of the original version of the *HelloWorld* program.

Why is this small Java program so confusing? Maybe the problem is that Java is not a pure object-oriented language and some of these features introduce confusion to most of us. As Cardelli noticed "*Java represents a healthy reaction to the complexity trend, but is more complex than many people realize.*" [Ca 96]

8.2. Comparison of MOOL and other OOL.

In this section we summarize the features of MOOL, comparing them with the features of other languages like Modula-3, Java, C#, and MOBY.

We have separated the comparison in several tables to make it more readable. Table 8.1 shows a comparison of features related to types. Table 8.2 shows a comparison of the statements provided by the languages. Table 8.3 shows a comparison of the features related with modules and genericity. Table 8.4 present a comparison of several other features that are present in these programming languages.

Feature	Modula-3	C#	Java	MOBY	MOOL
type annotations	yes	yes	yes	yes	yes
primitive types	integer subrange enumeration char boolean float	int short long byte sbyte uint ushort ulong float double decimal char bool struct enum	byte short int long char float double boolean	Bool Int Long Integer Float Double Extended Char String Exn Order enumeration	boolean integer float
automatic boxing / unboxing	no	yes	not yet (1.5)		yes
reference types	pointer object	pointer objects string class	class interface array objects	Ref Ptr object	class class interface object

Table 8.1. Features related to types.

Feature	Modula-3	C#	Java	MOBY	MOOL
assignment	yes	yes	yes	yes	yes
procedure call	yes	yes	static methods		yes
block	begin-end	{ }	{ }	{ }	{ }
exceptions	yes	yes	yes	yes	no
if	yes	yes	yes	yes	yes
case (switch)	yes	yes	yes	no	yes
loop (do)	yes	yes	yes	no	no
for	yes	yes	yes	yes	yes
while	yes	yes	yes	no	yes
repeat	yes		no (do while)	no	no
typecase	yes	no	no (instanceof)	no	no (instanceof)

Table 8.2. Statements.

	Feature	Modula-3	C#	Java	MOBY	MOOL
modules and module interfaces	modules	yes	no	no	yes	yes
	nested modules	no			yes	no
	interfaces / signatures	yes	no	no	yes	yes
	packages / namespace	(modules)	yes	yes	(modules)	(modules)
	nested pack/namespaces	no	yes	yes	(modules)	no
generics	modules	yes	no	no	yes	no
	module interface	yes	no	no		no
	functions	yes	no	no	yes	yes
	classes	no	not yet	not yet	yes	yes
	interfaces		not yet	not yet	yes	yes
	methods	no	not yet	not yet	yes	yes
	parameters allowed	all types	all types	references	all types	all types

Table 8.3. Features related with modules and genericity.

Feature	Modula-3	C#	Java	MOBY	MOOL
Static typechecking	yes	yes	yes	yes	yes
strongly typed	yes	yes	yes	yes	yes
implementation inheritance	single	single	single	single	single
interface inheritance		multiple	multiple	multiple	multiple
object-model	class	class	class	class	class
dynamic dispatch	single	single	single	single	single
Binding	static (procedures) dynamic (methods)	default static dynamic virtual	default dynamic static static	static (procedures) dynamic (methods)	static (procedures) dynamic (methods)
overloading	yes	yes	yes		yes
nested classes	no	yes	yes	no	no
garbage collection	yes	yes	yes	yes	yes
unified type system	no	yes	no	no?	yes
subtyping	structural	nominal	nominal	structural	nominal
class=type	yes/no(brand)	yes	yes	no	yes
parameter passing	value reference	value	value		value

Table 8.4. Other features.

8.3. Contributions.

In this section we enumerate our contributions more precisely. They are:

- The design of a new programming language that provides genericity, modules and object-oriented features.
- A model to include classes and modules in a programming language.
- A simple class mechanism that supports a minimal set of features that are inherently related to classes.
- A simple module system with two constructs: module interfaces and module bodies, which are used to encapsulate, hide information and code reuse.
- A mechanism to provide parameterized types (classes, interfaces and functions) to develop generic programming.
- A model to provide a unified type system where all types are derived from the same hierarchy of types.
- The definition of MOOL using an extended BNF grammar.

8.4. Future work.

There are many paths to follow to extend our research presented in this dissertation. Some of them are:

- Define the formal semantics of the language. An operational semantics can be defined and a translator can be implemented.
- The translator can generate an intermediate language that may be easily directed to Java bytecode or the MSIL of the .NET platform.
- Define a sound type system for MOOL to provide static typechecking to detect errors at compile time. This process reduces testing and debugging sessions.
- Extend the language with new features like exceptions, threads, mixins, or multiple dispatch with overloaded functions to implement binary methods.

References.

[ABC 03] Eric Allen, Jonathan Bannet, and Robert Cartwright. A First-Class Approach to Genericity. In *Proceedings of OOPSLA'03*, *Conference on Object-Oriented Programming, Systems, Languages and Applications*, Anaheim, California, October 2003.

[AC 96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer Verlag. 1996.

[ADA 80] United States Department of Defense. *Reference manual for the Ada programming language.* GPO 008-000-00354-8, 1980.

[AFM 97] Ole Agesen, Stephen Freund, and John C. Mitchell. Adding type parameterization to the Java™ Language. In *Proceedings of OOPSLA'97, Conference on Object-Oriented Programming, Systems, Languages and Applications*, Atlanta, Georgia, October 1997, pages 215-230.

[AG 98] Ken Arnold and James Gosling. *The Java™ Programming Language*. Addison Wesley. 1998.

[ALZ 03] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam - Designing a Java Extension with Mixins. In *ACM Transaction on Programming Languages and Systems (TOPLAS)* Volume 25, Issue 5, pages 641-712. September 2003. Previous version on Università di Genova, Dipartimento di Informatica e Scienze dell'Informazione, Technical Report DISI-TR-99-15. 1999.

[AZ 01] Davide Ancona and Elena Zucca. True Modules for Java-Like Languages. Università di Genova, Dipartimento di Informatica e Scienze dell'Informazione, Technical Report DISI-TR-00-12. In *Proceedings of ECOOP'01, European Conference on Object-Oriented Programming*, Budapest, Hungary, June 2001. Volume 2072 Lecture Notes in Computer Science, Springer, 2001.

[B 01] Joshua Bloch. *Effective Java™* Addison Wesley. 2001.

[B 02] Kim Bruce. *Foundations of Object-Oriented Languages Types and Semantics.* MIT-Press 2002.

[B 04] Gilad Bracha. Generics in the Java Programming Language. March 9, 2004. Available online at http://java.sun.com/j2se/1.5.0/lang.html

[B 92] Gilad Bracha. *The Programming Language JIGSAW: Mixins, Modularity, and Multiple Inheritance.* PhD Thesis. University of Utah, Salt Lake City, UT, USA 1992.

[BAF 03] Lujo Bauer, Andrew W. Appel, and Edward W. Felten. "Mechanisms for Secure Modular Programming in Java." In *Software--Practice and Experience* 33:461-480, 2003. Previous version in 1999.

[BC 90] Gilad Bracha and William Cook. Mixin-Based Inheritance. In *Proceedings of the Joint ACM OOPSLA '90, Conference in Object-Oriented Programming, Systems, Languages, and Applications and the European Conference on Object-Oriented Programming*, Ottawa, Canada, October 1990.

[BC 97] John Boyland and Giuseppe Castagna. Parasitic Methods: An Implementation of Multi-Methods for Java. In *Proceedings of OOPSLA'97, Conference on Object-Oriented Programming, Systems, Languages and Applications*, Atlanta, Georgia, October 1997.

[BCC+ 96] 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.

[BCK+ 01] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Max, Martin Odersky, Sven-Eric Paintz, David Stoutamire, Kresten Thorup, and Philip Wadler. Adding generics to the Java™ Programming Language: Participant Draft Specification. April 27, 2001. Available online at http://java.sun.com

[BCK+ 03] Gilad Bracha, Norman Cohen, Christian Kemper, Martin Odersky, David Stoutamire, Kresten Thorup, and Philip Wadler. Adding generics to the Java™ Programming Language: Public Draft Specification, Version 2.0. June 23, 2003. Available online at http://java.sun.com

[BD 98] Boris Bokowski and Markus Dahm. Poor Man's genericity for Java. In Clemens Cap, editor, *Proceedings JIT''98, Java-Informations-Tage*. Springer-Verlag, 1998.

[BDG+ 88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. *ACM SIGPLAN Notices*, Volume 23, Issue SI, September 1988.

[BFSG 03] Kim B. Bruce, Adrian Fiech, Angela Schuett, and Robert van Gent. PolyTOIL: A Type-Safe Polymorphic Object-Oriented Language. *ACM TOPLAS* volume 25, number 2, March 2003, pages 225-290. Early version appears in Proc. *ECOOP '95, European Conference on Object-Oriented Programming*, Aarhus, Denmark; August 1995 in Lecture Notes in Computer Science 952, Springer-Verlag, 1995.

[BG 03] Joshua Bloch and Neal Gafter. Forthcoming Java™ Programming Language Features. Presentation in JavaOne Conference. June 2003.

[BL 91] Gilad Bracha and Gary Lindsrom. Modularity meets inheritance. University of Utah, Technical Report UUCS-91-017. October 1991.

[BOSW 98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. GJ Specification. Manuscript, 1998. Available at the GJ web site. URL http://www.cs.bell-labs.com/who/wadler/pizza/gj/index.html

[BOSW 98a] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. GJ: Extending the Java Programming Language with type parameters. Manuscript, March 1998, revised August 1998.

[BOSW 98b] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language (GJ) In *Proceedings of OOPSLA '98, Conference on Object-Oriented Programming, Systems, Languages and Applications*, Vancouver, British Columbia, Canada. October 1998.

[BPF 97] Kim B. Bruce, L. Peterson, Adrian Fiech. Subtyping is not a good "Match" for object-oriented languages. In *Proceedings of FOOL '97 Workshop on the Foundations of Object-Oriented Languages*, Paris, France; 1997.

[BPV 98] Kim B. Bruce, Leaf Petersen, and Joseph Vanderwaart. Modules in LOOM: Classes are not enough. Manuscript. William College, 1998.

[BS 03] Maria Lucia Barron-Estrada and Ryan Stansifer. A Comparison of Generics in Java and C#. In Proceedings of ACM Southeast Regional Conference. Savanna, Georgia, April 2003.

[BS 03b] Maria Lucia Barron Estrada and Ryan Stansifer. Inheritance, Genericity and Binary Methods in Java. In *Computacion y Sistemas,* Volumen VII, numero 2. Mexico, DF. December 2003.

[C 95] Giuseppe Castagna. "Covariance and contravariance: Conflict without a cause." In *ACM Transactions on Programming Languages and Systems*, Volume 17, number 3, pages 431-447, May 1995.

[C 98] Craig Chambers and The Cecil Group. The Cecil Language Specification and Rationale. Version 3.0. Department of Computer Science and Engineering, University of Washington. December 9, 1998. Available online at http://www.cs.washington.edu/research/projects/cecil/www/cecil.html

[Ca 89] Luca Cardelli. Typefull programming. Research report 45, DEC Systems Research Center, Palo Alto, CA. May 1989.

[Ca 96] Luca Cardelli. Bad Engineering properties of Object-Oriented Languages. ACM Computing Surveys, volume 28 issue 4es, Article 150. December 1996. Special issue: position statements on strategic directions in computing research.

[Co 89] William Cook. A Proposal for Making Eiffel Type-Safe. In *Proceedings ECOOP '89*, *European Conference on Object-Oriented Programming*, Nottingham, UK, pages 57-70. July 1989. Cambridge University Press Cambridge 1989.

[C# 01] Standard ECMA-334. *C# Language Specification* [Online] http://www.ecma.ch December 2001.

[C# 02] Microsoft. New C# Language Features. White paper presented by Anders Hejlsberg at OOPSLA 2002. Available online at http://www.gotdotnet.com/team/csharp/learn/Future/default.aspx

[CCH+ 89] Peter Canning, William Cook, Walter Hill, John C. Mitchell, and Walter Olthoff. F-bounded quantification for object-oriented programming. In *Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[CGL 95] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. "A calculus for overloaded functions with subtyping." In *Information and Computation* 117,1,115-135 1995. A preliminary version in *Proceedings of the 1992 ACM Conference on LISP and Functional Programming* (San Francisco, June 1992)

[CHC 90] William R. Cook, Walter L. Hill, and Peter S Canning. Inheritance Is Not Subtyping. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 125-135, san Francisco, California, January 1990.

[CL 94] Craig Chambers and Gary T. Leavens. Typechecking and Modules for Multimethods. In *Proceedings of OOPSLA '94*, *Conference on Object-Oriented Programming, Systems, Languages and Applications,* pages 1-15, Portland, Oregon, October 1994.

[CL 97] Craig Chambers and Gary T. Leavens. BeCecil, a Core Object-Oriented Language with Block Structure and Multimethods: Semantic and Typing. In *Proceedings of Fourth International Workshop on Foundations of Object-Oriented Languages(FOOL 4)* Paris, France, January 1997. Early version on Technical Report #UW-CSE-96-12-02. December 1996. Available online at http://www.cs.washington.edu/research/projects/cecil/www/pubs/BeCecil.html

[CLCM 00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multi-Java: Modular Open Classes and Symmetric Multiple Dispatch for Java. Iowa State University. Technical Report #00-06a. April 2000

[CM 99] Craig Chambers, and Todd Millstein. Modular Statically Typed Multimethods. In *Proceedings of ECOOP '99*, *European Conference on Object-Oriented Programming*, Lisbon, Portugal, June 1999. Volume 1628 Lecture Notes in Computer Science, pages 279-303. Springer-Verlag Berlin Heidelberg 1999.

[CMP 00] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Developing Applications With Objective Caml.* O'REILLY, Paris, France. 2000.

[CS 98] Robert Cartwright and Guy L. Steele. Compatible Genericity with Run-Time Types for the Java™ Programming Language. (NextGen) In *Proceedings of OOPSLA '98*, *Conference on Object-Oriented Programming, Systems, Languages and Applications*, Vancouver, British Columbia, Canada. October 1998.

[CW 85] Luca Cardelli and Peter Wegner. "On understanding Types, Data Abstraction and Polymorphism." In *ACM Computing Surveys*, volume 17, number 4, pages 471-522, December 1985.

[DGLM 95] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proceedings of OOPSLA '95 Conference on Object-Oriented Programming, Systems, Languages and Applications* pages 156-158. Austin, Texas, October 1995.

[DN 81] Ole-Johan Dahl and Kristen Nygaard. "The Development of the {Simula} Languages." In *History of Programming Languages*, edited by Richard L. Wexelblat, pages 439-480. Academic Press, New York. 1981.

[EKMS 97] Mark Evered, James L. Keedy, Gisela Menger, and Axel Schmolitzky. Genja – A New Proposal for Parameterized Types in Java. In *Proceedings of Technology of Object-Oriented Languages and Systems*, 25, Melbourne, Australia, 1997.

[FF 98] Matthew Flatt, Matthias Felleisen. Units: Cool modules for HOT Languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*. 1998.

[FF 98b] Robert B. Findler and Matthew Flat. Modular Object-Oriented Programming with Units and Mixins. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98),* Baltimore, Maryland, USA, September 27-29, 1998.

[FKF 98] Matthew Flatt, Shriram Krishnamurthi, Matthias Felleisen. Classes and mixins. In *Proceedings of PoPL'98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, California. 1998.

[FR 99] Kathleen Fisher and John Reppy. Foundations for MOBY classes. Technical Memorandum, Bell Labs, Lucent Technologies, Murray Hill, NJ, February 1999.

[FR 99b] Kathleen Fisher and John Reppy. The design of a class mechanism for MOBY. In *Proceedings of ACM SIGPLAN'99 Conference of Programming Language Design and Implementation*, pages 37-49, Atlanta, Georgia, May 1-4, 1999.

[GJSB 00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification. Second Edition.* Addison Wesley, Boston Mass., 2000.

[H 87] Charles Antony Richard Hoare. "Hints on Programming Language Design." In *Programming Languages: A Grand Tour*, edited by Ellis Horowitz, Computer Science Press, 1987, pages 31-40. Reprinted from *SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, October 1973.

[I 78] Daniel H. Ingalls, "The Smalltalk-76 Programming System: Design and Implementation," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978, pages 9-16.

[JSR 014] Sun Microsystems. Adding Generic Types to the Java™ Programming Language. Java Specification Request JSR-000014, 1998. [Online] URL http://www.jcp.org/en/jsr/detail?id=14 Approved in 1999.

[JSR 201] Sun Microsystems. Extending the Java Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import. Java Specification Request JSR 201. Available online at http://jcp.org/en/jsr/detail?id=201

[KS 01] Andrew Kennedy and Don Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In Cindy Norris and James B. Fenwick Jr. editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Languages Design and Implementation* (PLDI 01), pages 1-12, New York, NY. June 2001. ACM Press. Appears as volume 35, number 5 of *SIGPLAN* Notices.

[L 00] Xavier Leroy. A modular module system. In *Journal of Functional Programming*, 10(3):269-303, 2000. Research Report 2866 INRIA April 1996.

[L 88] Henry Lieberman. Position statement for OOPSLA'88 Panel on Sharing Mechanisms. In *Proceedings OOPSLA '88, Conference on Object-Oriented Programming, Systems, Languages and Applications* page 359. In *ACM SIGPLAN* Notices, volume 23, issue 11, September 25-30, 1988.

[L 94] Xavier Leroy. Manifest types, modules and separate compilation. In *ACM Symposium on Principles of Programming Languages* 1994, pages 109-122. ACM Press, 1994.

[L+ 81] Barbara Liskov et al. *CLU Reference Manual.* LNCS 114 Springer Verlag, Berlin, 1981.

[LSU 87] Henry Lieberman, Lynn Andrea Stein, and David Ungar. The Treaty of Orlando. In Addendum to the *Proceedings of OOPSLA '87, Conference on Object-Oriented Programming, Systems, Languages and Applications, pages 43,44.* Orlando, Florida, October 1987.

[M 86] Bertrand Meyer. Genericity versus Inheritance. In *Proceedings of OOPSLA '86, Conference on Object-Oriented Programming, Systems, Languages and Applications,* pages 391-405. Portland, Oregon, USA. September 1986.

[M 92] Bertrand Meyer. *The Eiffel Language*. Prentice Hall. 1992.

[MBL 97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov Parameterized Types for Java (PolyJ). In *Proceedings 24th ACM Symposium on Principles of Programming Languages*, pages 132-145, Paris, France, January 1997.

[MFH 01] Sean McDirmid, Matthew Flatt, Wilson C. Hsieh Jiazzi: New-Age Components for Old-Fashioned Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '0).*pages 211-222. Tampa Bay, Florida, USA. October 2001.

[MFH 02] S. McDimid, M. Flatt, and W.C. Hsieh. Expressive modular linking for object-oriented languages. Technical report UUCS-02-014, 2002

[MMN 93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[MMS 79] J.G. Mitchell, W. Maybury, and R. Sweet. *Mesa Language Manual.* XEROX PARC CSL-79-3, April 1979.

[MTH 90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts. 1990.

[N 91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[OW 97] Martin Odersky, Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings 24th ACM Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, January 1997.

[P 01] Amit Jayant Patel. *OBSTACL: A language with objects, subtyping and classes.* PhD thesis, Stanford University. Stanford, California, USA. December 2001.

[P 02] Benjamin Pierce. *Types and Programming Languages.* The MIT Press, Cambridge, Massachusetts, 2002.

[P 72] David Lorge Parnas. A Technique for Software Module Specification with Examples. *Communications of the ACM*, Volume 15, Number 5, pages 330-336, May 1972.

[P 72b] David Lorge Parnas. On the Criteria To Be Used in Decomposing Systems Into Modules. *Communications of the ACM*, Volume 15, Number 12, pages 1053-1058, December 1972.

[PS 94] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems.* Willey 1994.

[R 02] 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.

[RV 98] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension of ML. In *Theory and Practice of Object Systems*, 4(1):27-50, 1998.

[S 91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991.

[S 92] 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.

[S 97] Andrew Shalit. The Dylan Reference Manual. 1997. HTML version available at http://www.gwydiondylan.org/drm/drm_1.htm

[T 93] Antero Taivalsaari. *A Critical View of Inheritance and Reusability in Object-Oriented Programming.* PhD Thesis. University of Jyväskylä, Finland, December 1993.

[T 97] Kresten Krab Thorup. Genericity in Java with Virtual Types. In *Proceedings of ECOOP'97, 11th European Conference on Object-Oriented Programming,* Jyväskylä, Finland, June 1997.Volume 1241 of Lecture Notes in Computer Science, pages 444–471, Springer-Verlag, Berlin Heidelberg 1997.

[US 87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings OOPSLA '87, Conference on Object-Oriented Programming, Systems, Languages and Applications.* In *ACM SIGPLAN* Notices, pages 227-242, December 1987.

[U 88] David Ungar. Are classes obsolete?. A panelist perspective. In *Proceedings OOPSLA '88, Conference on Object-Oriented Programming, Systems, Languages and Applications* page 358. In

ACM SIGPLAN Notices, volume 23, issue 11, September 25-30, 1988.

[V 01b] Mirko Viroli. Parametric polymorphism in Java: an Efficient Implementation for Parametric Methods. In *Proceedings* of *SAC' 01,* 16th *ACM Symposium on Applied Computing,* Las Vegas, Nevada, USA, March 2001, pages 610-619.

[W 01] Ralph Westfall. Hello, World Considered Harmful. Technical opinion in Communications of the ACM. Volume 44, number 10, pages 129-130. October 2001.

[W 02] Conrad Weisert. Pseudo Object-Oriented Programming Considered Harmful. ACM SIGPLAN Notices. Volume 37 (4) April 2002.

[W 83] Niklaus Wirth. *Programming in Modula-2.* Texts and Monographs in Computer Science, Springer-Verlag 1983.

[W 87] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings OOPSLA '87, Conference on Object-Oriented Programming, Systems, Languages and Applications.* In *ACM SIGPLAN* Notices, pages 168-182, December 1987.

[W 87b] Niklaus Wirth. "On the Design of Programming Languages." In *Programming Languages A Grand Tour*, edited by Ellis Horowitz, pages 23-30. Computer Science Press, 1987. Reprinted from IFIP Congress 74, 386-393, North-Holland, Amsterdam, North Holland Publishing Company.

[WS 03] Mario Wolczko and Randall B. Smith. Prototype-Based Application Construction Using SELF 4.0. Available online at http://research.sun.com/research/self/release_4.0/Self-4.0/Tutorial/index.html Downloaded in December 2003.

[XB 03] Cong-cong Xing and Boumediene Belkhouche. On Pseudo Object-Oriented Programming Considered Harmful. Technical opinion in Communications of the ACM. Volume 46, number 10, pages 115-117. October 2003.

[YKS 04] Dachuan Yu, Andrew Kennedy, and Don Syme. Formalization of Generics for the .NET Common Language Runtime. In *Proceedings 31th ACM Symposium on Principles of Programming Languages*, pages, Venice, Italy, January 2004.

Appendix.

The Grammar of MOOL.

This section presents the grammar for MOOL. This grammar uses the following BNF conventions:

[x] Denotes zero or one occurrence of x.

{ x } Denotes zero or more occurrences of x.

x | y Denotes one of either option x or y.

x y Denotes sequence, x followed by y.

x & y Denotes x or y or x y.

( ) Used to group elements.

Terminal symbols are represented in **bold** font.

Non-terminal symbols are represented in the LHS followed by **::=**.

Basic productions.

AccessModifier	::= **protected**
BooleanLiteral	::= **true** | **false**
NullLiteral	::= **null**
Digit	::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
Digits	::= Digit {Digits}
Number	::= Digits | Digits**.** Digits
Letter	::=**a** |**b** |**c** |**d** |**e** |**f** |**g** |**h** |**i** |**j** |**k** |**l** |**m** |**n** |**o** |**p** |**q** |**r** |**s** |**t** |**u** |**v** |**w** |**x** |**y** |**z** | **A** |**B** |**C** |**D** |**E** |**F** |**G** |**H** |**I** |**J** |**K** |**L** |**M** |**N** |**O** |**P** |**Q** |**R** |**S** |**T** |**U** |**V** |**W** |**X** |**Y** |**Z**
OtherChar	::= BlankSpace |**.** |**,** |**:** |**;** |**<** |**>** |**/** |**** |**[** |**]** |**{** |**}** |**!** |**%** |**&** |***** |**(** |**)** |**_** |**+** | | | **-** | **=**
CharSequence	::= Letter | Digit | OtherChar
Comments	::= **//** {CharSequence}
Identifier	::= Letter { Letter | Digit }
Literal	::= Number | BooleanLiteral |NullLiteral
QualifiedIdentifier	::= Identifier [**.** Identifier]
ParameterizedIdentifier	::= Identifier **<**TypeList**>**
TypeName	::= **object** | Identifier [**.** Identifier]| ParameterizedIdentifier
IdentifierList	::= Identifier {**,** Identifier }
BasicType	::= **integer** | **float** | **boolean**
Type	::= (BasicType | TypeName) { **[ ]** }
TypeList	::= Type { **,** Type}

Compilation unit productions

ProgramUnit	::= ModuleInterface | ModuleImplementation
ModuleInterface	::= **module interface** Identifier ModuleBlock
ModuleImplementation	::= **module** Identifier ModuleInterfaces ModuleBlock

ModuleBlock ::= **{** {ImportDeclaration} {ModuleDeclarations} [ModuleInit] **}**

ModuleDeclarations ::= ConstantDeclaration | GenericFunctionDeclaration | TypesDeclaration | VariableDeclaration

ModuleInterfaces ::= **implements** IdentifierList **;**

ImportDeclaration ::= **import** Identifier [.Identifier] [as Identifier]**;**

ModuleInit ::= **init** Block

ConstantDeclaration ::= **const** Type Identifier **=** ConstExpression **;**

GenericFunctionDeclaration ::= (Type | **void**) Identifier [TypeParameters] FormalParameters (**;** | Block)

TypesDeclaration ::= ClassDeclaration | InterfaceDeclaration | FunctionType

VariableDeclaration ::= Type Identifier [**=** Expression] **;**

FormalParameters ::= **(** [Parameters] **)**

Parameters ::= Parameter {**,** Parameter}

Parameter ::= Type Identifier

Type productions

ClassDeclaration ::= **class** Identifier [TypeParameters] [Superclass] Interfaces ClassBodyDec

InterfaceDeclaration ::= **class interface** Identifier [TypeParameters] [ExtendsInterfaces] InterfaceBodyDec

FunctionType ::= **function** (Type | **void**) Identifier FormalParameters **;**

SuperClass ::= **extends** IdentifierTypeParameter

Interfaces ::= **implements** InterfacesList

ExtendsInterfaces ::= **extends** InterfacesList

ClassBodyDec ::= **{** {ClassVariables} [FieldsList] ConstructorsList [MethodsList] **}**

InterfaceBodyDec ::= **;** | **{** { FunctionMethodDeclaration } **}**

InterfacesList ::= IdentifierTypeParameter {**,** IdentifierTypeParameter}

IdentifierTypeParameter ::= Identifier [TypeParameters]

TypeParameters ::= **<** TypeParameterList **>**

TypeParameterList ::= TypeParameter {**,** TypeParameter}

TypeParameter ::= TypeVariable Bounds

TypeVariable ::= Identifier

Bounds ::= [SuperClass & Interfaces]

ClassVariables ::= VariableDeclaration

FieldsList ::= **fields** FieldDeclaration { FieldDeclaration}

FieldDeclaration ::= [**shadow**] Type Identifier = Expression **;**

ConstructorsList ::= **constructors** ConstructorDeclaration
{ConstructorDeclaration}

ConstructorDeclaration ::= Identifier FormalParameters (Block | ;)

MethodsList ::= **methods** MethodDeclaration {MethodDeclaration}

MethodDeclaration ::= [**override | shadow**] FunctionMethodDeclaration

FunctionMethodDeclaration::= [AccessModifier] (Type | **void**) Identifier FormalParameters
(Block | ;)

Block ::= **{** {LocalVariableDeclaration} Statements **}**

LocalVariableDeclaration ::= Type Identifier [**=** Expression] **;**

Statements productions

Statements ::= [Statement { **;** Statement}]

Statement ::= AssignmentStatement | Block | BreakStatement |
ContinueStatement | ExpressionStatement | ForStatement |
CallStatement | IfStatement | SwitchStatement |
ReturnStatement | WhileStatement |;

AssignmentStatement ::= Expression AssignOperator Expression

BreakStatement ::= **break**

ContinueStatement ::= **continue**

ElseClause ::= **else** Statement

ExpressionStatement ::= Expression

ForStatement ::= **for (** [ForInit] **;** [Expression] **;** [ForUpdate] **)** Statement

CallStatement ::= Expression **(** ActualParameters **)**

IfStatement ::= **if (**Expression**) {** Statement [**;** ElseClause] **}**

ReturnStatement ::= **return** [Expression]

SwitchStatement ::= **switch (** Expression **)** SwitchBlock

WhileStatement ::= **while (**Expression**)** Statement

ActualParameters ::= [Expression { **,** Expression }]

Case ::= **case** ConstList **:** Statement

ConstList ::= Literal {, Literal}

DefaultStatement ::= **default :** Statement

Designator ::= Identifier

ForInit ::= Type Identifier = Expression

ForUpdate ::= Identifier = Expression

SwitchBlock ::= **{** {Case} DefaultStatement **}**

Expression productions

ConstExpression ::= Expression

Expression ::= AndExpression { OrOperator AndExpression}

AndExpression ::= NegExpression {AndOperator NegExpression}

NegExpression ::= {NotOperator} RelExpression

RelExpression ::= AddExpression {RelOperator AddExpression}

AddExpression ::= MultExpression {AddOperator MultExpression}

MultExpression ::= UnaryExpression {MultOperator UnaryExpression}
[InstanceExpression]

UnaryExpression ::= (**++**|**--**)UnaryExpression {**+**|**-**}UnaryExpression |
OtherUnaryExpression

InstanceExpression ::= UnaryExpression **instanceof** Type

OtherUnaryExpression ::= PrimaryExpression {Selector}{**++**|**--**} | ~ UnaryExpression | CastExpression

PrimaryExpression ::= Identifier | Literal | **(**Expression**)** | **this** [**(** ActualParameters **)**] | **super** SuperSuffix | **new** Expresion

Selector ::= ArrayAccess | **(** ActualParameters**)** | **.**Expression

ArrayAccess ::= **[** Expression **]** {**[** Expression **]** }

CastExpression ::= **(** Expression | Type **)** UnaryExpression

SuperSuffix ::= **(** ActualParameters **)** | .Identifier [**(** ActualParameters **)**]

AssignOperator ::= **=**

RelationalOperator ::= **==** | **!=** | **>=** | **<=** | **>** | **<**

OrOperator ::= **||**

AndOperator ::= &&

NotOperator ::= !

AddOperator ::= + | -

MultOperator ::= * | / | %