Implementing Leda: Objects and Classes

Jim Shur
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3202

91-60-11

Implementing Leda: Objects and Classes

Jim Shur
Department of Computer Science
Oregon State University
Corvallis, OR
97331
shurj@mist.cs.orst.edu

November 22, 1991

Abstract

Leda is a strongly typed, compiled, multiparadigm programming language. This paper describes various implementation concerns which arose from the experience of writing a Leda compiler as part of the Leda research team. These include aspects of run-time representation, symbol-table information, and code generation. The paper concentrates on *objects* and *classes*. An overview of the object-oriented features of the language is given, including our semantic view of *parameterized classes*.

1 Introduction

Leda is a multi-paradigm, strongly typed, compiled programming language. The paradigms supported are procedural, functional, relational, and object-oriented. The primary purpose of the language, according to its designer, "is to provide a vehicle for experiments in multi-paradigm programming" [Bud89b]. Using relational programming techniques in Leda was originally described in [Bud89a]. Further ideas on that topic—the language definition is still evolving—can be found in [Bud91b]. First class functions and functional programming are discussed in [Bud89b]. The object-oriented paradigm along with the data structures that support its use are presented in [Bud89c]. A more recent paper, [Bud91c], describes a style of programming which combines the different paradigms.

In January of 1990, the Leda research team began the project of implementing a Leda compiler. At the time of this writing, the compiler is generating 68000 assembly language object programs for two different computers. More information on the actual compiler is given in Appendix A. This paper reports on the experience of that implementation, specifically those parts relating to the object-oriented features. First we discuss Leda in the light of past and current research relating to multiparadigm languages. Section 3 gives a brief introduction to the language with some examples. Section 4 explains the run-time representation of objects, an issue which had to be resolved before a compiler could be built. The system for gathering symbol information is presented in Section 5. Finally, Section 6 addresses code generation.

2 Related Work

"Which programming language am I going to use?" asks E. W. Dijkstra in the preface of *A Discipline of Programming*[Dij76]. A recent list names approximately 1300 programming languages from which the eminent computer scientist might choose [Lan91]. Dijkstra's belief that a programming language—whether we like it or not—influences our thinking habits, elevates the importance of the choice beyond aesthetic considerations. It seems that our approach to a given problem is born out of our intellect, associations, and the tools we find ourself faced with. It is not surprising that the old adage, *the right tool for the right job*, applies to programming languages. Unfortunately programmers are faced with many and varied jobs over time. Worse yet, even a single job may be quickly decomposed into disparate sub-tasks. Provided with a monistic approach to problem-solving, a programmer can be expected to feel no less frustration than a carpenter forced to make do with only a screwdriver. An area of research is born: What is the best way to make sure that a programmer has a rich set of tools so that diverse problems can each be met in a natural, straightforward way?

Each of the three major first-generation programming languages were well suited for particular tasks. Around 1966, the second-generation language PL/1 was developed as a synthesis of these—Fortran, Cobol, and Algol 60 [Weg76]—in hopes that the language would be a good tool for scientific, business, systems, and combined applications. This shows one way to attack the problem—a single language providing multiple tools. Whether that approach is preferable to providing several specialized languages within some integrated environment is an open question. The latter solution is called *mixed language programming* in [E&G84], and the authors suggest that "in many applications, various parts of a complete program are best written in assorted languages." The paper cites the difficulties of providing a suitable interface between modules written in the different languages. An advantage of this approach is that one can still make use of existing code, e.g. numerical subroutines written in Fortran. Current research in mixed language programming is discussed in [H&S90]. This paper gives evidence that "it is often possible to find more common ground between disparate programming languages and models than might be expected." This is the attitude of the Leda research team, though we apply it to the single-language approach. That is, we believe that we can exploit the commonality of different languages to create a single one which allows the programmer to apply one of several models of computation to a given problem.

In the 25 years or so since the advent of PL/1, researches have made significant progress in finding unique problem-solving approaches, or computational models. Because some of these are so distinct and single-focused, forcing one to make a near radical shift in thinking to move from one approach to another, they have come to be known as *paradigms*. Recognized paradigms we will consider here include procedural, functional, logical, and object-oriented. Whereas PL/1 undertook to combine different linguistic *features*, they were all within the context of the procedural paradigm. At this point in history the problem of providing a state-of-the-art toolbox to the programmer becomes more difficult as different paradigms must be made available. Thus recent research in this area has come to use the terms *multiparadigm systems*, which may include *multiparadigm languages*, and/or *multiparadigm environments*.

The importance of this research is argued in [Bob84]. The author criticizes the sole use of the logical paradigm (in the form of Prolog) as the basis for fifth-generation programming. He admits its power but asserts that "no single paradigm is appropriate to all problems, and powerful systems must allow multiple styles." The same author with some colleagues discuss the multiparadigm programming language Loops in [S&B86]. This language takes a lesser known paradigm called *access-oriented* and combines it with the better known paradigms mentioned above. A high priority in the development of Leda is the study of the interaction between different paradigms. Our research should be made easier by keeping with those paradigms which are well studied and worked out, at least individually.

Multiparadigm systems are treated apart from any particular paradigm in [Hai86a], which introduces two criteria the author believes are necessary for a multiparadigm system:

- A multiparadigm system should allow language elements from different paradigms to co-exist within one program or module

- Each paradigm of such a system should be able to refer to and depend upon services provided by the other paradigms

Our philosophy coincides with these points; Leda meets these guidelines.

Some researchers have worked to reap the benefits of combining paradigms not by studying multiparadigm systems as such, but by extending an established language of one paradigm with features of another. Following the example of adding classes to Algol 60 to create Simula [B&D73], Bjarne Stroustrup created C++ by adding object-oriented features to C [E&S90]. Both of these extensions allow a mixture of procedural and object-oriented programming. Object-oriented programming and functional programming are combined in the Common Lisp Object System (CLOS), an object-oriented extension to Common Lisp [Ste90]. Yet another combination, functional and logic programming, are combined in a system described in [K&E88]. The authors ask how to combine the two paradigms so that the best features of each are preserved. Our research with Leda concerns an abstraction of that same question. The [K&E88] solution, unlike the extended languages above, is to leave versions of Prolog and Lisp basically intact and construct an interface bridge between them. The authors state an advantage to this approach is that there is no degradation of performance due to one language being implemented on top of another. They admit however, that the programmer would be living in "two separate worlds, each with its own name spaces and syntactical rules." Our decision with Leda has been to concentrate on the human side first—ease of programming, comfort in moving between the different paradigms—relegating efficiency concerns to important-but-secondary status.

It is interesting that some multiparadigm langauges have at their heart some simple unifying model that belie their many-sided exteriors. The language Nial, with roots in APL, has the *nested array* as its sole data structuring capability [J&G86]. The development of Nial was motivated by the "desire to provide a multiparadigm programming language suitable for teaching the various styles." Although Leda was conceived as a general purpose language, its potential use in the classroom encourages us. In fact, Leda syntax has already been used in an upper-division course on programming languages to explain higher-order functions without having to stray too far from the familiar Pascal-like program structure. The language G, described in [Pla91], is a multiparadigm language which utilizes the *stream* as its fundamental data type. In addition to discussing multiparadigm research

```
qs := func(s)[local[x], x:=@s, if(x)[self(s[<x]), x, self([>=x])]].
```

Figure 1: The quicksort algorithm in the multiparadigm language G

in general, the author gives an example of a quicksort algorithm coded in G, shown in figure 1. The @ operator causes its argument to enumerate the initial value in its value sequence and then to move on to the next value in that sequence. self refers to the function being defined and is used here to make recursive calls. Although two languages may support the same basic paradigms, choices of data structures and syntax can make the languages quite distinct.

The language Orient84/K unifies the paradigms it supports in the object framework [Hai86b]. As explained in later sections, Leda too uses objects as a unified low-level representation. Orient84/K is still different in that objects consist of not only a behavior part, but also a knowledge-base part which can include Prolog-like rules and facts. Thus the object-oriented and logical paradigms are more finely integrated than in Leda.

By now the extensive range of research in multiparadigm systems should be apparent, and still much has gone unmentioned. Even the specialized-domain language for the *Mathmatica* software package supports functional, object-oriented, and rule-based programming [Wol88]. In his overview of the subject, Brent Hailpern speculates "that many more iterations of the experiment/theory cycle will come before this area is mature [Hai86a]." Our goal is for Leda to be one more iteration toward that end.

3 The Language

This section begins with a brief introduction to programming in Leda, especially the use of classes and their objects, in an attempt to impart some of the flavor of the language. It concludes with a more detailed discussion of the use and semantics of parameterized types. For more information, especially in regard to mixing the various paradigms, we refer the reader to the papers mentioned in the introduction. Figure 2 gives a skeleton of a Leda program. The strong resemblance to Pascal is not accidental. A likely pitfall in designing a multiparadigm language is assumed to be the creation of an overly complex affair susceptible to such appellations as "kitchen sink," [Bud91c] "swiss army knife," or worse yet—"fatal disease."[1]. Hence a high priority design goal of Leda is to retain simplicity, while still providing a tool of many dimensions. A legal Leda program need only consist of a single compound-statement containing at least 1 (possibly empty) statement. Figure 3 (a) shows the smallest valid Leda program. Part (b) of that figure shows a simple counting program which makes use of one of the standard control structures provided. **While**, **repeat**, and **for** loops may be used for iteration; **if-then** along with **if-then-else** constructs are available for selection.

The program in part (c) of Figure 3 defines a new class Point with data variables x and y. A method distance gives objects of the class the ability to compute their distance from some other point passed as an argument. Class definitions consist of two sections of variable declarations. First come the *instance* variables which are unique for every object which is an instance of the class. Next,

[1] Dijkstra's epithet for the programming language PL/1 [Dij72]

```
const
  // constant declarations (the "//" marks the rest of a line as a comment)
type
  // type declarations, class definitions
var
  // variable declarations

  // function and method definitions
begin
  // program statements
end;
```

Figure 2: Skeleton of a Leda program

following the keyword **shared**, are *shared* variables, existing in singular form, independent of any particular object—shared by all instances of the class. Shared variables may be accessed through an object or through the class itself. For example p1.distance and Point.distance are aliases, both referring to the method distance in class Point. They are not interchangeable within any Leda expression however. Invoking the method by accessing it through the class name, as in Point.distance(), is illegal since to actually invoke the method an object is needed to act as receiver. The first assignment below is illegal as well.

```
p1.distance := function(q1, q2 : Point)->real;  // WRONG!
               begin
                 // ...
               end;

Point.distance := function ... // Correct
```

When assigning to a shared variable the class name must be used to access the member. We hope that this rule will keep the programmer and readers of the program aware that such an assignment will affect *all* instances of the class. The incorrect example above may lead one to mistakenly assume that only the state of object p1 is being altered.

All classes understand the message new and respond to it by dynamically creating a new instance of themselves. This is how objects are born. Although not fully implemented, the intent of Leda is to automatically discard objects in those cases where it can be determined that they are no longer referenced by any variables, guaranteed to remain thenceforth unused. Variables begin their lives in a formal state of being *undefined*. They can be released from that condition by assigning to them either a constant (certain pre-defined classes only), a defined variable, or a newly created object. A special built-in polymorphic predicate defined will take any object as an argument and return a boolean indicating its circumstance. If during the course of a running Leda program, an expression attempts to access an instance member via an undefined variable, a run-time error occurs. Unfortunately it is impossible to check for this situation at compile time in the general case. Shared members are more

```
(a)  begin
       ;      // an empty statement
     end;

(b)  var
       i : integer;
     begin
       for i := 1 to 10
         print(i);
     end;

(c)  type
       Point := class
           x : real;
           y : real;
         shared
           distance : method(Point)->real;
       end;
     var
       p1 : Point;
       r  : real;

     method Point.distance(P : Point)->real;
     begin
       return sqrt(((P.x - x) * (P.x - x)) + ((P.y - y) * (P.y - y)))
     end;

     begin
       p1 := Point.new();                // create the Point (0,4)
       p1.x := 0;                        //   using the message 'new'
       p1.y := 4;
       r  := p1.distance(Point(3,0));    // ask p1 how far it is from (3,0)
                                           using the constructor for Point
       r.print();                        // prints 5
     end;
```

Figure 3: (a) The smallest valid Leda program; (b) A program that counts to 10; (c) Defining a class, creating new objects, sending a message

robust and *can* be accessed through undefined variables. This may seem surprising. Normally shared members are accessed through a class pointer which is part of the object at run-time. This allows for dynamic binding, the use of the shared member associated with the class of the actual object being held by the variable at run-time, not necessarily the staticly defined class of the variable. When a variable is undefined, the shared member can not be accessed in this way. The compiler generates code to check for this situation. If the variable is found to be undefined, the shared member is taken directly from the statically defined class. This includes messages which are implemented by shared methods. They may be sent to an undefined variable which is the receiver. Control is passed to the method and it is the method's responsibility to manage the possibility of an undefined receiver or not. Possible actions could be to substitute a default value, or print out or return an error condition. Note that the receiver is always passed by value so defining it within the method is not an option; the receiver will remain undefined when control returns from the method.

All classes have the ability to be invoked as a subprogram which is the *constructor* for the class. The constructor must be given an argument for each instance member inherited by, or explicitly defined within the class. The order of the parameters must be the same as the order defined, starting with the inherited members. The constructor sends the new message to the class to create a new object, and then assigns each parameter to its respective instance variable. The new object is returned. The keyword NIL may be used as a parameter where the programmer wishes the corresponding instance member of the new object to remain undefined. Use of the constructor for class Point can be seen in Figure 3 (c).

Leda supports the basic tools of object-oriented programming—subclassing, inheritance, virtual methods, overriding, and dynamic binding. Not only methods can be virtual, all declarations in the shared portion of a class are automatically treated as virtual and can be overridden in a descendant class. Objects may be assigned to variables declared to be of the same or any ancestor class. The actual shared member accessed by the dot operator depends on the class of the actual object referenced by the variable at run-time, not on the declared class of the variable (unless, as explained above, the variable is undefined).

3.1 Parameterized Classes

Besides instance and shared variables, a class definition may introduce new types which are local to the scope of the class definition. These we call *type parameters* and are declared immediately following the class keyword enclosed in parenthesis. Classes with one or more type parameters are called *parameterized classes.* The definition of these types is severely restricted so that they may be used to implement genericity as discussed below. There are two ways to declare a local class within a class. The first, exemplified by the type parameter T of the class Pair shown in Figure 4 (a), creates the most minimal class possible— one with neither instance nor shared variables. Though slight, such a type is far from useless, at least when taking a view from inside the class within which it is locally defined. As the swap method shows, some computations only require that an object of some type be assigned an object of the same type; the particular properties of the type are irrelevant. Similarly, variables of a type parameter can be passed as parameters to a subprogram expecting that same type, or returned from a function declared to return the type. The second sort of in-class local type declaration is one which defines a superclass for the type. Although no *additional* instance nor shared variables may be declared, the inheritance mechanism works as usual so that variables declared to be of the type parameter can safely access any of the inherited members. Figure 4 (b) gives an example. Type parameters declared to have a superclass are called *constrained* type parameters; the superclass is the *constraining type*. Otherwise, the type parameter is *unconstrained.*

The motivation for allowing these local type declarations within a class is as a means to implement genericity. On their own, the type parameters don't add power or expressiveness to the language since they are local to some class and therefore incompatible with any types defined outside the class. The significance of a parameterized class from *outside* the class is completely different from the inside view described above, giving type parameters a sort of dual semantics. A parameterized class is actually an implicit definition of 1 or more (possibly infinite) classes which take the place of the literal parameterized definition written by the Leda programmer. The programmer is then free to make use of these classes in type and variable declarations. The classes that are created implicitly are those that can be constructed by substituting some class from within the current class hierarchy for each occurrence of the type parameter within the parameterized class. If the type parameter is unconstrained, there is no restriction on which classes may be substituted. Constrained type parameters may only be substituted by the constraining class itself or one of its descendants. The implicitly defined types are denoted by the name of the parameterized class followed by the substitution classes in parenthesis as arguments. When a programmer refers to an implicitly defined class in this manner we say that the parameterized class is being *instantiated.* Instantiated types may also be used as substitution types as long there is no constraining type to prevent it. This is how a parameterized class definition can give rise to an infinite number of implicitly defined classes. For example a class List defined to have an unconstrained type parameter T, defines the class List(integer), List(List(integer)), List(List(List(integer))), and so on. Figures 5 and 6 give examples of complete programs which use parameterized classes.

The implicitly defined classes are themselves arranged hierarchically. If foo is a parameterized class, then foo($p_1, p_2, \ldots, p_n$) is a descendant of foo($q_1, q_2, \ldots, q_n$) if and only if each of the p parameters is identical to or a descendant of its corresponding q parameter. This notion makes sense intuitively if one ponders assigning a List(Dog) to a List(Animal) where Dog has been defined as a subclass of Animal. But examination uncovers a very different sort of class hierarchy than the one we are familiar with when parameterized classes and genericity are not involved. Consider the List class from Figure 6 in conjunction with an Animal class that has a subclass Dog. The class List(Animal) defines an instance variable first to be of class Animal. The class List(Dog) defines that same instance variable first to be of class Dog. This is a case of *strengthening,* or *restricting,* the type of an inherited instance variable to a more specific class, something that would never be allowed when defining a subclass in a traditional class system. Fortunately the type restriction is what we want. It is this feature that makes genericity useful and expressive by allowing us to abstract out the common attributes of the family of List classes, making use of polymorphic code while still exerting control over the sorts of objects that may be referred to by class members. We can guarantee a List(Dog) will only contain Dogs. Had we relied solely on the inheritance mechanism, we would have been forced to explicitly define each List class separately, duplicating members and methods along the way.

The cost of allowing type strengthening within the implicitly defined class hierarchy is the extra care that must be taken to insure type safety with regard to Leda's strong typing. One consequence is that shared variables may not be declared with a type parameter as its class. Another is the

```
(a) type
      Pair := class (T)
          first  : T;
          second : T;
        shared
          swap : method();
      end;

      method Pair.swap();
      var
        temp : T;
      begin
        temp := first;
        first := second;
        second := temp;
      end;

(b) type
      hasFirst := class
          first : integer;
        shared
          getFirst : method()->integer;  // returns member first (not shown)
      end;

      foo := class (T < hasFirst)  // local type T is subclass of hasFirst
          bar : T;
        shared
          firstBar : method()->integer;
      end;

      method hasFirst.getFirst()->integer;
      begin
        return first;
      end;

      method foo.firstBar()->integer;
      begin
        return bar.getFirst();   // getFirst is inherited from hasFirst
      end;
```

Figure 4: (a) Declaring a class within a class and its use in a method; (b) A constrained type parameter

```
type
  // In class curry, the do method is a curry of binary function f, fixing the
  //   second parameter to y.  The class may be used with any function by
  //   instantiating the type parameters T, U, & V to the respective argument
  //   and return types.
  curry := class(T, U, V)
             y : U;
             f : function(T, U)->V;
           shared
             do : method(T)->V;
           end;

  point := class
      x, y : real;
    shared
      distance : method(point)->real;
  end;
  intCurry    := curry(integer, integer, integer);
  mixedCurry  := curry(point, point, real);
var
  plus3 : intCurry;
  fromOrigin : mixedCurry;
  i : integer;
  r : real;

  method curry.do(x : T)->V;
  begin
    return f(x, y);
  end;

  method point.distance(P : point)->real;
  var
    r : real;
   begin
     r := ((P.x - x) * (P.x - x)) + ((P.y - y) * (P.y - y));
     return r.sqrt();
   end;
begin
  plus3 := intCurry(3, integer.plus);
  i := plus3.do(7);    i.print();  // prints 10
  i := plus3.do(12);   i.print();  // prints 15
  fromOrigin := mixedCurry.new(point(0,0), point.distance);
  r := fromOrigin.do(point(3,4));   r.print();  // prints  5
  r := fromOrigin.do(point(5,12));  r.print();  // prints 13
end;
```

Figure 5: Using parameterized classes—the curry example

```
type
  list := class (T)
            first : T;
            rest  : list(T);
          shared
            append : method(T);
          end;

  intList  := list(integer);
  realList := list(real);

var
   i : intList;
   r : realList;

method list.append(next : T);
type
  Tlist := list(T);
begin
  if (defined(rest)) then
    rest.append(next)
  else
    begin
      rest := Tlist.new(next, NIL);
    end;
end;

begin
  i := intList.new(10, NIL);
  i.append(9);
  i.append(8);
  i.append(7);

  r := realList.new(0.1, NIL);
  r.append(0.01);
  r.append(0.001);
  r.append(0.0001);
end;
```

Figure 6: Using parameterized classes—the list example

inability for a class, defined to be a subclass of an instantiated type, to override any of the methods originally defined in the parameterized class. [Bud91a] explains the inability to ensure type-safe behavior in these situations.

4 Objects and Run-Time Representation

The underlying model of Leda centers on objects, classes, and messages. The language represents entities as *objects* which are in turn, *instances* of *classes*, in the sense introduced by the language Simula 67. An early decision in our implementation was to generalize this representation to all data types, including basic types such as integers, boolean values, functions, and even classes. Accordingly, operations on these basic types are actually *methods* defined within their respective classes, so that an expression such as $3 + 4$ is a *message* to the integer object 3 to add itself to the integer 4 and return the result. The decision was motivated by the wish to generate truly polymorphic code for the methods of parameterized classes. That is, one block of code that will work correctly for any instance of any class defined by instantiating the same parameterized class. This of course requires generating code for objects of unknown type, making it necessary that all objects have the same size. Our particular solution to the uniformity problem was inspired by the experience of the Smalltalk implementors who write in [G&R89] that:

> The contention that even the addition of two integers should be interpreted as message sending met with a certain amount of resistance in the early days of Smalltalk. Experience has demonstrated that the benefits of this extreme uniformity in the programming language outweigh any inconvenience in its implementation. Over several versions of Smalltalk, implementation techniques have been developed to reduce the message-sending overhead for the most common arithmetic operations so that there is now almost no cost for the benefits of uniformity. [2]

The following sections show how objects are represented internally at run-time. Section 4.1 gives the basic structure of a generic object. Section 4.2 shows how some of the traditional data types provided by Leda are implemented as classes, and what their objects look like. Finally, Section 4.3 explains how classes are implemented as objects, and their run-time connection with those objects that are instances of them.

4.1 Internal structure of an object

All objects in Leda are pointers. An object points to an *instance table*. The instance table is a data structure which contains the instance variables of the object, that is, the variables defined in the class of the object to be unique to each object. Each object has its own personal copy of the instance variables. The instance table also contains a pointer to the object's *shared table*. This data structure contains the shared variables—those variables defined in the class of the object to be shared among, or common to, all instances of the class. Finally, the instance table contains a reference count. As a result of Leda's pointer semantics, more than one variable may denote the same object. The reference count keeps track of how many such variables there are. If the reference count is zero, the space taken up by the instance table can be returned to free storage.

[2] [G&R89], page 119

The shared table also contains some fields other than the shared variables themselves. These include the object which is the class in which the shared variables are defined, and the object which is the superclass of that class. These fields can be used at run-time to access information about the dynamic class of an object and its place within the class hierarchy. (Presently only one such function is available to the Leda programmer. By sending the message filter to a class with one argument, the class will either return the argument itself if it is an instance of the class or one of its subclasses, or otherwise return undefined. Future research may utilize these fields for other purposes, such as to give objects the ability to"clone" themselves for example.) To be precise, the shared table does not contain the shared variable objects, but pointers to them. The necessity for this is explained in [Bud91a]. Figure 7 shows the run-time representation of an object in Leda.

4.2 Representing traditional data types as objects

Leda provides programmers with the predefined types integer, real, function, method, and boolean. The implementation treats methods as a special kind of function. At run-time, the two types are indistinguishable; they are represented the same way, and methods are considered to be objects of the class function. Thus methods will not be discussed further in this section. Details on the implementation of methods vs. functions can be found in [Che91]. The classes corresponding to the predefined types are on an equal footing with any classes the Leda programmer may define. What makes them different is that they may contain instance or shared variables of type primitive. Variables of this type may not presently be manipulated within the language. Thus the methods for these predefined classes are written directly in assembly language, in a separate module which is linked with a Leda source program at compile time. Figure 8 shows the integer class in pseudo-Leda syntax, the integer 7 as respresented in assembly code, and the code for the plus method. Note that all the operations on integers—arithmetic and relational—are defined as methods within the class. Since the variable value is a primitive, we can't actually define the class and its methods within Leda, so they are included in the special assembly language module. The class real is implemented similarly.

Functions in Leda are objects with two instance variables—a pointer to the code, and a pointer to the environment of definition. The environment pointer actually points to the activation record of the subprogram in which it is defined. Access to non-local variables in Leda is implemented using static chains. When a function is invoked, the environment variable from the instance table is used for the static link. The class function, along with the code generated for a function definition is shown in Figure 9.

Enumerated types, of which the predefined type boolean is one example, are also implemented as classes. Enumerated types are central to the use of the relational paradigm within Leda. Unlike languages such as Pascal and C, it is necessary, for the sake of meaningful output, to maintain at run-time the literal strings corresponding to the different enumerated constants. This is achieved by creating a class for each enumerated type that is defined in a Leda program. The instance variable for the class is a primitive integer value, which can be efficiently manipulated in functions such as successor and predecessor. The shared variables for the class include the number of constants defined for the type, and a table of strings. The value field of an instance of an enumerated type can by used as an index into the table of strings by any methods (such as print) defined for the class. Further details on implementing enumerated types in Leda can be found in [Pes91].

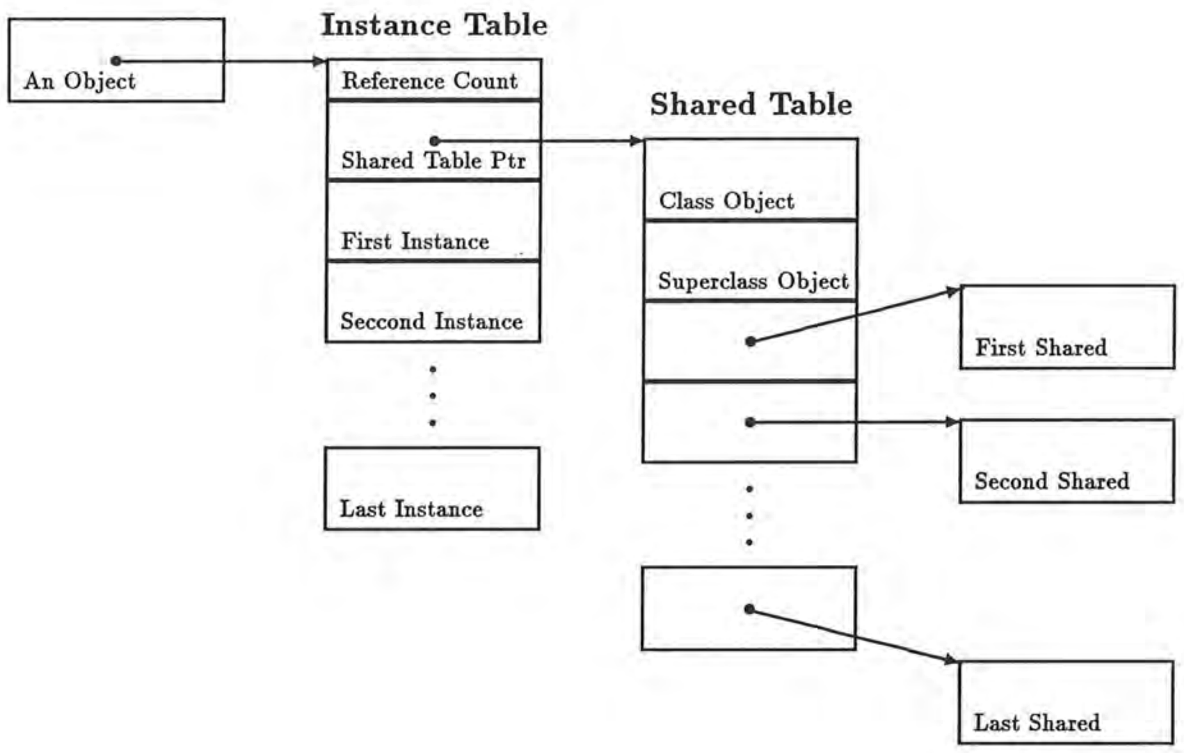

Figure 7: An object in Leda

```
integer := class
    value : primitive;
  shared
    plus : method(integer)->integer;
    minus : method(integer)->integer;
    times : method(integer)->integer;
    slash : method(integer)->integer;
    mod : method(integer)->integer;
    unaryPlus : method()->integer;
    unaryMinus : method()->integer;
    print : method(integer);
    less, lessEqual : method(integer)->boolean;
    greater, greaterEqual : method(integer)->boolean;
    equal, notEqual : method(integer)->boolean;
end;
```

```
IC_7:                                | The object 7
                .long   IC_7_inst    | Pointer to instance table
IC_7_inst:
                .word   1            | reference count
                .long   C1_shared    | Pointer to shared table
                .long   7            | value
```

```
|===========================================================================
C1_plus_code:                             | Code for method integer.plus
|---------------------------------------------------------------------------
                link    a6,#-0            | no locals
                movl    a6@(12),a1        | receiver object in a1
                movl    a1@(6),d1         | receiver integer value in d1
                movl    a6@(16),a1        | argument object in a1
                movl    a1@(6),d2         | argument integer value in d2
                addl    d2,d1             | computer sum
                movl    d1,sp@-           | save sum value on stack
                movl    #10,sp@-          | push size on stack
                jsr     _malloc           | create space for new integer
                movl    d0,a0             | put new object in a0
                addql   #4,sp             | pop size off stack
                movl    sp@+,d1           | restore saved value
                movw    #0,a0@(0)         | initialize reference count
                pea     C1_shared         |
                movl    sp@+,a0@(2)       | load pointer to shared table
                movl    d1,a0@(6)         | load new value
                bra     epilog2           | clean-up
```

Figure 8: The integer class, the object 7, and the method integer.plus

```
function := class
    code_pointer : primitive;
    environment_pointer : primitive;
end;
```

```
var
  f : function();
begin
  f := function();
       begin
         // function statements
       end;
end;
```

```
F0:
       | ... (code for the function statements)
F_end0:         .data               | lay out the function object
F_obj0:                             | the function object itself
                .long   F_inst0     | pointer to instance table
F_inst0:        .word   0           | ref. count (will be 1 after assignment)
                .long   C2_shared   | pointer to class function's shared table
                .long   F0          | pointer to the code
F_env0:         .long   0           | pointer to the environment
                .text               | resume executable statements
                movl    a4,F_env0   | place frame pointer (a4) in environment
       | ... (complete assignment to f)
```

Figure 9: The **function** class, a **Leda** function definition, and the corresponding generated code including the function object

4.3 Classes are objects too

Classes are objects, each of which is an instance of a corresponding metaclass. A class object for each class defined in a Leda source program is laid out in the object program. Individual metaclasses do not need a physical manifestation however, their existence is more for the sake of semantics. All instances of metaclasses are linked to a single common shared table, which contains the new method. This method allows classes to dynamically create new instances of themselves. The instance variables for a class object include the information necessary for the new method—the size of an instance of the class, and the location of the shared table for the class. Like function objects, class objects have a pointer to their code, which is the class constructor, and a pointer to the environment for the constructor. The other instance variables are exactly those objects which are shared variables from the point of view of instances of the class. This bears repeating: The shared variables for an instance of some class A, are the instance variables for the object which *is* class A. This relationship can be implemented very neatly. Recall that the shared table contains pointers to the shared variables. In fact, the pointers in the shared table point right back to the instance variables in the corresponding class object. Figure 10 shows a class defined in Leda with a picture of the class object and its relationship to an instance of the class. Figure 11 shows the code for the new method which will dynamically create a new instance of any class.

Besides the aesthetic appeal of a unified model, implementing classes as objects have a practical and simplifying value. To avoid confusing Leda programmers and readers of Leda programs, it was decided that when assigning a value to a shared variable of a class, the class name must be used. An assignment to a shared variable accessed through an instance variable might not make it apparent enough that the change will affect *all* instances of the class. With the class as an object, an assignment such as

```
Circle.area := function(x : circle)->real; begin ... end;
```

requires no special code generation techniques. Circle.area can be treated as any object and its member variable—used as the target of an assignment statement, or in an expression.

Intuitively one might correctly assume that when altering a shared variable of a class, not only all instances of that class are affected, but all instances of any class which inherits the particular shared variable are affected as well. It is much less intuitive that a change in a shared variable of a class would affect instances of ancestor classes *from which* the shared variable is inherited. For this reason Leda only allows referencing shared variables using *the class of definition*. This refers to the class in which the shared variable is originally defined or a class in which the variable is explicitly overridden. Since a class object only includes as instance variables the shared variables originally defined or overridden in the class definition, the rule is enforced through the natural means of checking membership. Figure 12 shows the class layouts for a class and its subclass. The reader will note that many labels in the assembly code begin with a capital *C* followed by a number. Every class defined in a Leda program is given a unique number by the compiler. This number can then be employed to generate unique labels, so that a print method defined in a class foo will not clash with a print method defined in some class bar for example.

```
type
  Circle := class
      center : Point;
      radius : real;
    shared
      area : method()->real;
      circumference : method()->real;
      distance : method(Circle)->real;
  end;
var
  c : circle;
begin
  c := circle.new();
end;
```

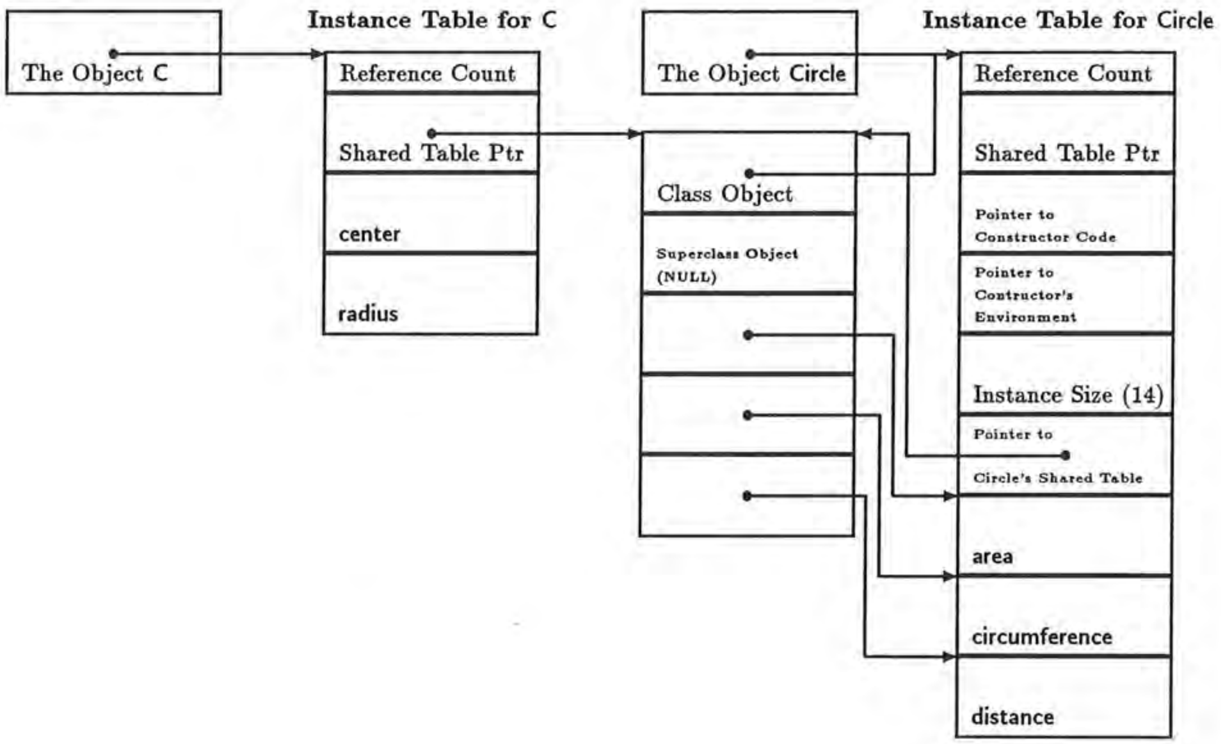

Figure 10: A class definition, the class, and an instance

```
|============================================================================
C0_new_code:    link    a6,#-0          | create new instance of receiver
|----------------------------------------------------------------------------
movl a6@(12),a1 | receiver in a1
movl a1@(14),sp@- | push obj size on stack
jsr _malloc | get space for new object
movl d0,a0 | put new obj in a0
addql #4,sp |
movl 0,a0@(0) | init reference count
movl a6@(12),a1 | return receiver in a1
movl a1@(18),a0@(2) | init shared table
bra epilog1 | clean-up
```

Figure 11: The new method in class metaclass

5 Symbol Information

Having decided on the run-time representation of objects and the idea that all types would be implemented as classes, all entities would be objects, and all operators message-sends, the compiler could be built. This section discusses our method of collecting the information from the type and variable declarations that is necessary for type checking and code generation. After showing where the symbol information is kept in relation to the overall structure of the compiler, we present a major component of Leda's symbol information system, the classID. Then we discuss one particualr feature of the classID—the class offset table—in more detail.

We developed the Leda compiler using the compiler construction tools flex and bison, (compatible with the lex and yacc programs from the unix operating system), in conjunction with the programming language C++. We approached the problem of compilation from an object-oriented point of view. The compiler works in two stages. The first phase reads through the Leda source program and constructs a heterogeneous tree structure (an abstract syntax tree of sorts) where each node is an object representing a declaration, statement, expression, or some other component. At this time the compiler collects symbol information, and performs type checking and closure analysis. The latter two tasks, along with the necessity to perform closure analysis in a separate phase than one which performs code generation, are described in [Che91]. The second phase consists of a message to the tree to generate code. The message is received and passed on to child nodes who deal with the message as they see fit. Figure 13 shows the main classes in the implementation that deal with symbol information.

Symbols are those entities which occur on the left hand side of some declaration. These may be type, variable, constant, argument, or type parameter declarations. Argument and type declarations are always in the context of some subprogram definition. Type parameters always relate to the definition of a class in a subprogram's Type section. Variable declarations may occur in either of these two places. Thus another look at Figure 13 shows how all the information is originally

```
Point := class
    x, y : integer;
  shared
    distance : method(Point)->real;
end;

Circle := class of Point
    radius : real;
  shared
    area   : method()->real;
    circum : method()->real;
end;
```

```
C12:                                    | the class Point
                .long   C12_inst        | pointer to inst table
C12_inst:                               | instance table for Point
                .word   1               | reference count
                .long   C0_shared       | ptr to metaclass shared table
                .long   C12_constr_code | ptr to constructor
                .long   0               | environment for constructor
                .long   14              | size of an instance of class Point
                .long   C12_shared      | location of Point' shared table
C12_distance:   .long   0               | the shared variable - distance
C12_shared:                             | Point's shared table
                .long   C12_inst        | the class Point
                .long   0               | no superclass
                .long   C12_distance    | pointer to first shared variable

C13:                                    | the class Circle
                .long   C13_inst        | pointer to inst table
C13_inst:                               | instance table for Circle
                .word   1               | reference count
                .long   C0_shared       | ptr to metaclass shared table
                .long   C13_constr_code | ptr to constructor
                .long   0               | environment for constructor
                .long   18              | size of an instance of class Circle
                .long   C13_shared      | location of Circle's shared table
C13_area:       .long   0               | the shared variable - area
C13_circum:     .long   0               | the shared variable - circum
C13_shared:                             | Circle's shared table
                .long   C13_inst        | the class Circle
                .long   C12_inst        | the class Point
                .long   C12_distance    | ptr to inherited var - distance
                .long   C13_area        | ptr to shared var - area
                .long   C13_circum      | ptr to shared var - circum
```

Figure 12: A superclass and its subclass laid out in memory

```
class programASTnode : public scope { // Head node for a subprogram
protected:
  DTnode  *args;              // formal arguments
  DTnode  *constantdefs;      // 'const' declarations
  DTnode  *variabledefs;      // 'var' declarations
  DTnode  *typedefs;          // 'type' declarations
  // other state variables
public:
  // methods
};

class classType : public scope { predefined + user defined classes
  offsetTableNode *offsets;
public:
  // methods
};

class userclassType : public classType { // classes created by the programmer
protected:
  declTypeID  *paramTypes;
  typeVarNode *superClass;
  declVarID   *instanceVars;
  declVarID   *sharedVars;
public:
  // methods
};

class DTnode { // base class for type, variable, and arg. declarations
protected:
  char      *name;      // Left hand side of declaration
  ASTnode   *type;      // Right hand side of declaration
  classID   *classID;   // class name, structure, and args for type-checking
  bool      constInd;   // indicator for constant
  DTnode    *next;      // pointer to next declaration
public:
  // methods
};
```

Figure 13: Major C++ classes in symbol information system

collected within the objects corresponding to the above-mentioned scopes. The class programASTnode represents the root node of a subprogram. Objects of that class contain lists of constant, type, variable, and argument declarations, along with a list of statements that make up the code of the subprogram. Of course one of the expressions within the statement list may itself be a subprogram and so the nesting of subprograms in Leda is captured by the recursive structure of the syntax tree. An object of class userclassType is built for each class defined within the Leda source program. These objects contain the instance and shared variable declarations, the superclass linking the class to its inherited variables, and a list of type parameters. The type parameters are stored as a list of type declarations, reflecting the idea that they are classes defined within the scope of a class. The type field of a type declaration for a type parameter is itself an instance of class userclassType with null pointers for the instance and shared variables, and a superclass which is the constraining type, if present.

After this information is collected in raw form it is distilled into a form to facilitate type checking and code generation. On the subprogram level, classIDs are built for each type and variable declaration. For each pre- and user-defined class, a classID is built for each type parameter, and an *offset table* is built which contains the important information about the instance and shared variables, including those inherited from parent and ancestor classes.

5.1 The classID

The classID is a structure used in the implementation of Leda. Its purpose is to hold enough information about the type of a variable to be able to perform both type checking and code generation. This section describes the structure of the classID itself, and how a classID is provided for each type declaration of the Leda source program (including the types local to class definitions which are declared via type parameters). The assignment of a classID for each built-in and user-defined type is complicated by the fact that types can be interdependent, as well as directly or mutually recursive. Also note that Leda does not require forward declarations and types may be declared in any order as long as all names used in the type on the right-hand side of the declaration lie within the current name space. classIDs are assigned to each declaration "up front," as soon as each type section is parsed. Since the types associated with variable declarations can only be made up of class names that may be found as the left hand side of type declarations, by providing the classIDs for each such declaration, all necessary type information can be retrieved for any variable with minimum effort. How the classID is actually utilized for type checking and code generation is explained in [Che91]. The relevant C++ classes from the Leda compiler are given in Figure 14. As can be seen, the classID consists of a class name, a class structure, and a list of classIDs representing the arguments. Each component is discussed in the following subsections.

5.1.1 The class name

The class name always comes from the left hand side of the type declaration. Its purpose is to enable the type checker to compare type names along with the structural information. This allows the compiler to issue a warning if the names don't match for two types which are otherwise compatible. It is an open question exactly when the Leda compiler will give such warnings. Take for example,

```
type
```

```
class classIDlist {
  protected:
    classID     *first;
    classIDlist *next;
  public:
    // methods
};

class classID {
  protected:
    char         *className;
    classType    *classStructure;
    classIDlist  *classArgs;  // args used to instantiate parameterized classes
  public:
    // methods
};
```

Figure 14: C++ class definitions relating to the classID

```
   yards := integer;   // className field of classID is "yards"
   meters := integer;  // className field of classID is "meters"
```

As will be seen below, the classID contains enough information to let the type checker know that yards and meters are both aliases for the class integer. Consequently they are compatible in the sense that there is no danger of a run-time type error due to a difference in protocol. Including the names in the classID allows the type checker to give what appears to be, in this case, a pertinent warning when the types are mixed.

5.1.2 The class structure

The class structure contains much information about the type being declared. In all cases the class structure contains a unique class number which is assigned to all predefined classes as well as any user-defined classes. Also present is an offset table for code generation. The offset tables contain all the class member names including those that are inherited. Other information depends on the type of class structure. Functions and methods contain lists of argument types, a return type, and, for methods only, a receiver type. User defined classes include type parameters, instance and shared variables, and the immediate superclass. The classes from the Leda compiler which may be used as class structures include userclassType shown in Figure 13, and those shown in Figure 15.

If the type field (the right-hand side) of a declaration is itself a class structure, then that object is assigned to the classStructure field of the type's classID. The only other class of object that could be held in the type field of a declaration is a *type variable*. Type variables are used to create an alias for some other type which must be found in the name field of some declaration in the current name space.

```
class intType : public classType {
public:
  // methods
};

class realType : public classType {
public:
  // methods
};

class funcType : public classType {
private:
  typeArgsNode *params;
  ASTnode      *returnType;
public:
  // methods
};

class methType : public classType {
private:
  ASTnode      *receiverType;
  typeArgsNode *params;
  ASTnode      *returnType;
public:
  // methods
};
```

Figure 15: Some *class structures* from the Leda compiler

```
type
  intFun := function(integer)->integer;
  xFun   := intFun;  // xFun is an alias for intFun
```

In this case, the class structure for xFun is obtained by looking up the class structure of intFun. Thus the classIDs for intFun and xFun will have different names but identical class structures. Likewise, the declaration

```
  moreFun := xFun;
```

will have still a different class name but the same class structure.

When a type variable names a parameterized class, it must contains arguments in order to instantiate that class.

```
type
  pclass := class(T, U)
            ...
            end;

  foo := pclass(integer, real);
```

In the example above, the process of obtaining the class structure for foo's classID involves the extra step of instantiation. Instantiation is simply a textual substitution of the actual for the formal arguments in the class structure from the pclass declaration. In addition, the instantiated structure no longer has any type parameters.

5.1.3 The class arguments

Two type variables are compatible if their names and corresponding arguments are all compatible. Thus it is important not to lose the information about the arguments of a type variable when it is aliased. For this reason the classID contains within it a list of classIDs corresponding to these arguments if present. For instance, the classID for the type foo above will contain classIDs for the integer and real classes in its argument list.

5.2 Building the offset tables

Every time a classID is obtained its structure is checked for the presence of an offset table. If necessary, the offset table is built. To build the offset table, the compiler first gets the offset table from the structure's superclass. The superclass must be a type variable but it is important to note that it is not necessary to obtain the classID of the superclass including the arguments in order to get the offset table. What is done is to get the classID from the generic class represented by the superclass name (not including the arguments), and then instantiating the offset table with the arguments. The offset table is completed by adding any additional class members to the inherited ones. By not requiring the classIDs for the superclass arguments the system is significantly more flexible as seen below.

5.3 Filling the type section with classIDs

The methods given above for putting the classID into a given type declaration show the limits of recursion in type declarations, as well as the necessity to obtain the classIDs in a particular order. These points are discussed below.

5.3.1 Recursion

If the right-hand side of the declaration is a type variable, the name and all of the arguments must already have classIDs in order to build the classID for the declaration. This excludes recursive definitions (either direct or indirect) involving type variables.

```
type
  bar := list(bar);      // illegal, could never build a classID

  foo := gak(integer);  // mutual recursion, also illegal
  gak := list(foo);
```

Another dependency occurs when the right-hand side is a user-defined class containing a superclass. In order to build the offset table, the classID of the superclass name (not including the arguments as explained above) must be available. Thus no combination of type variables and superclass names can be directly or indirectly recursive.

```
type
  foo := class(T) of bar   // Illegal, foo depends on bar,
         ...               //   bar on gak, gak on foo
         end;
  bar := class of gak
         ...
         end;
  gak := foo(integer);


  foo := class(T) of bar   // This is OK because bar depends only
         ...               //   on list, not its argument, gak
         end;
  bar := class of list(gak)
         ...
         end;
  gak := foo(integer);
```

5.3.2 Getting things in order

After a declaration section is recognized by the parser, a message is sent to the Leda subprogram which "owns" those declarations, requesting that the type declarations be filled with classIDs. The method is shown in Figure 16 below. Upon receiving the message to fill the classIDs, the subprogram

```
void programASTnode::fillClassDefs()
{
  // get classIDs for the type declarations

  int IDsObtained;

  if (typedefs) {
    // fill the classIDs until unable to obtain any more
    do {
      IDsObtained = 0;
      IDsObtained = typedefs->fillClassDefs();
    } while (IDsObtained > 0);

    // fill the class IDs for type parameters within the classID structures
    typedefs->fillTypeParamDefs();

    // report on names with no defs
    typedefs->checkClassDefs();
  }
}

int DTnode::fillClassDefs(int count = 0)
{
  if (!classID && type->hasClassID()) {
    classID = type->getClassID();
    classID->setClassName(name);
    ++count;
  }

  if (next)
    return next->fillClassDefs(count);
  else
    return count;
}
```

Figure 16: Getting classIDs for type declarations

sends the message on to the list of type declarations. Each type declaration decides in turn whether or not it can obtain its classID by checking if all the types that it depends on have already obtained their classIDs. If so, the classID is filled and the message is sent down the line. The number of new classIDs successfully obtained is returned to the subprogram. If the return value is positive, the subprogram repeats the message to the list of type declarations in the hope that those declarations which were unable to obtain classIDs during the previous pass will have better luck with the now-larger set of types that have acquired their classIDs. As soon as the list of type declarations reports that no new classIDs were obtained, the subprogram knows that it is fruitless to continue; all classIDs that can be gotten have been gotten.

5.3.3 classIDs for type parameters

At this point a message is sent to the type declarations to look into the class structure within their classID, and give classIDs to the type parameters if any. Type parameters are stored as local-to-the-class type declarations. Parameterized type constraints are implemented as superclasses of the corresponding type parameters. Given that, the procedure for filling the classIDs is exactly the same as for the type declarations on the subprogram level.

Note that because classIDs for type parameters are obtained only after the classIDs for the user-defined classes, it is not possible to use a type parameter as a superclass name (shown below).

```
type
  foo := class(T) of T   // illegal, classID of foo depends on T
         ...
         end;
```

Finally, a message to check the classIDs is given to the type declarations and error messages are issued for any declaration with an empty classID.

5.3.4 Class scope

The definition of a class introduces a new scope beneath the scope of the subprogram in which the class is being declared. The scope includes the type parameters, which from the point of view of the class members and methods are viable types, and the instance and shared variable declarations. A subtle issue is determining at exactly what point in the definition of a class is the new scope activated. The answer is at the point after the type parameters and before the superclass. Thus the superclass, as well as of course the class member declarations, can refer to the type parameters, while a parameterized type restriction could not. (Recall from above that the superclass name itself may not be a type parameter however).

```
type
  U := boolean;

  foo := class(T)
         ...
         end;
```

```
class offsetTableNode {
private:
  char            *name;
  memberType      instORshared;
  int             classNum;
  ASTnode         *type;
  int             count;
  int             metacount;
  offsetTableNode *next;
public:
  \\ methods
};
```

Figure 17: Implementation of offset tables in the Leda compiler

```
  bar := class(U) of foo(U)     // argument U of foo is type param U
         ...
         end;
  gak := class(U < foo(U))      // argument U of foo is alias for boolean
         ...
         end;
```

5.4 More on Offset Tables

Every object in the compiler representing a class definition from a Leda source program contains an offset table. This data structure contains information about each field that may be accessed with the membership (dot) operator by an instance of the class. This includes all instance and shared variables defined in or inherited by the class, as well as updated information for shared variables which are overridden by the class definition. Offset tables are implemented by the class offsetTableNode which is shown in Figure 17.

The name field contains the name of the class member and the type field its type. instORshared marks whether the variable is defined in the instance or shared portion of the class. The classNum is a field which keeps track of the member's class of definition. This allows the code generator to construct the proper label in the class's shared table. The label must point to the appropriate shared object which, as explained above, is located in the class object where it was originally defined or last overridden. The count field determines the member's relative position within its relevant—instance or shared—run-time table.

5.4.1 The metaclass offsets

Before explaining the metacount field, we must note the dual existence of an offset table. Because the userclassType class contains all the information needed for the defined class as well as its metaclass, we chose not to use a seperate object to implement the metaclass. Thus a class's offset table must be prepared to respond to messages meant for the class as well as messages destined for the metaclass. The metacount is all that is necessary. This field gives the relative position of the member from the point of view of the metaclass. Specifically it is the location of the member within the instance table of the class object. For class members which aren't relevant to the metaclass, namely all instance variables and shared variables not defined nor overridden in the class, the metacount contains the special flag value -1 which makes the field "invisible" to offset table methods which act on behalf of the metaclass. A pair of Leda class definitions with their offset tables can be seen in Figure 18. Recall that all objects in Leda are the same size (the size of a pointer) so that the size of objects need not be stored, only their order.

The offset table is one of the key pieces of information included within the classStructure of the classID. The connection between the two go further, in that the offset tables are built for a userType, if necessary, whenever a classID is requested. To build an offset table, the object representing the Leda class must first obtain the offset table from the superclass if there is one. Recall that the classID, hence the offset table, of the superclass is guaranteed to be obtainable because of the careful order in which the classIDs are filled. If the superclass has any type arguments, the superclass's offset table is instantiated. Now the object is ready to add its own member variables to the offset table. A message is sent to the lists of instance and shared declarations telling them to add themselves to the offset table. Each declaration sends a message to the offset table with the pertinent information to add. The offset table then checks to see if the member name already exists. If it does, and it is an instance variable, then an error is reported since they may not be overridden. If it is a shared variable, and the class of definition is different, then the offset table entry is changed to admit the new class of definition. This ensures that both overridden and inherited class members maintain the same offset within the shared variable table as in their parent and ancestor classes. This feature is critical to allowing dynamic binding in particular, hence the object-oriented style of programming in general.

For those members which are newly defined in the class, a count is maintained as they get passed down the line of offset-table nodes so that when they get appended to the end they will have the proper counts and metacounts. Figure 19 shows the classType method which builds the offset table. Figure 20 contains the other methods related to that task.

5.5 Scoping

Although Leda is not a fully block-structured language in the sense of Algol-60, (declarations cannot be made on the compound-statement level), functions and methods may be nested to any practical degree. Thus it is not enough to merely store information about a symbol, retrieving it upon encountering the symbol in the code section of some Leda subprogram. Each scope may have its own declaration for a given symbol and the compiler must always match symbols with the corresponding information from the appropriate scope. The problem reduces to one of searching the various deposits of symbol information in the correct order, so that the first declaration found for a symbol is the right one. The Leda compiler tries to find the declaration for a symbol first in the local subprogram in which the symbol occurs. If not found, and the subprogram is a method, the compiler next searches the declarations of the class in which the method is defined. The next step is to check for the symbol within the outer subprogram in which the original subprogram is

```
type
  Part := class
      controlNum : integer;
      price : real;
      quantity : integer;
    shared
      print : method(); // print part information
      cost : method(integer)->real;  // calc cost of n parts
      fillOrder : method(integer)->integer;  // adjust quantity
  end;

  tipType := (phillips, slot);

  Screwdriver := class of Part
     style : tipType;
     length : real;
     gauge : real;
   shared
     compatible : method(Screw)->boolean; // Can screwdriver be used with screw?
     print : method(); // override the print method
   end;
```

class Part:

				meta-	
Name	IorS	classNum	count	count	type
controlNum	inst	12	0	-1	integer
price	inst	12	1	-1	real
quantity	inst	12	2	-1	integer
print	shar	12	0	0	method()
cost	shar	12	1	1	method(integer)->real
fillOrder	shar	12	2	2	method(integer)->integer

class Screwdriver:

				meta-	
Name	IorS	classNum	count	count	type
ControlNum	inst	12	0	-1	integer
price	inst	12	1	-1	real
quantity	inst	12	2	-1	integer
print	shar	13	0	0	method()
cost	shar	12	1	-1	method(integer)->real
fillOrder	shar	12	2	-1	method(integer)->integer
style	inst	13	3	-1	tipType
length	inst	13	4	-1	real
gauge	inst	13	5	-1	real
compatible	shar	13	3	1	method(Screw)->boolean

Figure 18: Two Classes and their offset tables

```
classID *userclassType::getClassID()
{
  offsetTableNode *classOffsets = NULL;
  typeArgsNode *tArgs;
  classType *ct;

  // build offset table if it hasn't been built yet
  if (!offsets) {
    setParentLevelDeepest();       // Have this scope point to the outer
    staticNesting->append(this);  //   and put it on the scope chain
    if (superClass) {
      ct = superClass->getClassDef();  // Get offset table of superclass
      classOffsets = ct->getOffsetTable();
      if (tArgs = superClass->getTypeArgs())
        // if the superclass has arguments instantiate it
        classOffsets = classOffsets->instantiate(tArgs, ct->getParamTypes());
      else
        // otherwise just copy it as is
        classOffsets = classOffsets->copy();
    }
    if (instanceVars)            // add the variables of this class to
      classOffsets =             //     the offset table
        instanceVars->addTOinstOffsets(classOffsets, uniqueScopeNum);
    if (sharedVars)
      classOffsets =
        sharedVars->addTOsharedOffsets(classOffsets, uniqueScopeNum);

    // If there is no superclass, instance, or shared vars, then need to create
    //   a dummy offset table node so that offsets isn't NULL.
    offsets = (classOffsets) ? classOffsets : new offsetTableNode();
    staticNesting->remove();  // return scope chain to previous state
  }
  // more code
}
```

Figure 19: Building the offset table

```
offsetTableNode *
    DTnode::addTOinstOffsets(offsetTableNode *offTab, int classNum)
{
  if (offTab)
    offTab->append(name, INST, classNum, type, 0, 0);
  else
    offTab = new offsetTableNode(name, INST, classNum, type, 0, -1);
  return ((next) ? next->addTOinstOffsets(offTab, classNum) : offTab);
}

offsetTableNode *
    DTnode::addTOsharedOffsets(offsetTableNode *offTab, int classNum)
{
  if (offTab)
    offTab->append(name, SHAR, classNum, type, 0, 0);
  else
    offTab = new offsetTableNode(name, SHAR, classNum, type, 0, 0);
  return ((next) ? next->addTOsharedOffsets(offTab, classNum) : offTab);
}

void  offsetTableNode::append(char *n, memberType iORs,
                              int i, ASTnode *t, int c, int mc)
{
  if (iORs == instORshared)  c++;
  if (i == classNum && instORshared == SHAR)  mc++;

  if (!(strcmp(n, name))) {
    if (iORs == INST || instORshared == INST) {
      yyerror("redefinition of instance variable ", name);
    }
    if (i == classNum) {
        yyerror("Error: redefinition of shared variable ", name,
                " in same class");
    }
    classNum = i;
    metacount = mc;
  }
  else {
    if (next)
      next->append(n, iORs, i, t, c, (iORs == SHAR ? mc : -1));
    else
      next = new offsetTableNode(n, iORs, i, t, c, (iORs == SHAR ? mc : -1));
  }
}
```

Figure 20: Auxiliary methods for building the offset tables

```
class scope : public ASTnode {
protected:
  int uniqueScopeNum;  // allows unique reference to subprograms and classes
  scope *parentLevel;  // points to next outer scope
public:
  static int scopeNumCounter;          // keeps track of last unique number used
  static scope *deepest;               // stores current local scope
  static void removeDeepestLevel() {  // removes (pops) local scope
    deepest = deepest->parentLevel;
  }
  void addToScope() {                  // adds (pushes) new local scope
    parentLevel = deepest;
    deepest = this;
  }
  // other methods
};
```

Figure 21: Implementing scoping with the abstract superclass scope

defined. This process continues, searching outer-nested subprograms followed by class definitions when methods are involved, until finally the highest (global) level scope is reached. If the symbol remains unfound then an error is reported.

The scoping system in the Leda compiler is implemented with the class scope (shown in Figure 21) which acts as a stack of naming environments. scope is an abstract superclass whose two children are the classes classType and programASTnode, corresponding to classes and subprograms respectively, previously shown in Figure 13. scope makes use of a static data member deepest to keep track of the current local scope (the most deeply nested) which of course changes as the compiler works its way through a Leda source program. As the methods for class scope show, subprograms and classes have the ability to make themselves the current scope. The scope class itself uses the static method removeDeepestLevel to discard the most deeply nested scope in favor of its parent level. Figure 22 shows part of the method which implements the message to subprograms requesting them to generate code for themselves, demonstrating the adding and removal of scopes. Note that every scope contains a pointer to its parent environment which is set automatically when it becomes the new local scope. When a scope is unable to satisfy a message to return information for a given symbol, it checks to see if it has a parent level and if so, passes the message to it.

It is not always the case that the search for symbol information should begin at the most deeply nested level in the local context. Let's look at the example in Figure 23. To check the legality of the assignment in line 15, the compiler must get information for the symbol f which, by beginning at the local scope, it finds to be of class foo. From there it is determined that f.a is of class bar. Now the compiler needs to get information about the symbol bar—but wait! If the search were to begin at the local scope, it would determine that bar was an alias for boolean and the assignment would be disallowed. This is incorrect since at the time (and scope) that a was declared to be a bar,

```
ASTnode *programASTnode::genCode()  // Note: brief outline of actual method
{
  // if the subprogram has a receiver add a scope
  //  for the class
  if (receiver) {
    classType *receiverType =
      receiver->getType()->getClassID()->getClassStructure();
    receiverType->addToScope();
  }
  // now add the scope for the subprogram itself
  this->addToScope();

  // if types are defined, lay out the class objects etc.
  // if there are statements, have them generate code

  if (typedefs) typedefs->genCode();
  if (statements) statements->genCode();

  // remove the scope (or scopes) that were added
  if (receiver) scope::removeDeepestLevel();
  scope::removeDeepestLevel();
}
```

Figure 22: Adding and removing scopes

```
type
  bar := integer;
  foo := class
           a : bar;
         end;

  function fun(x : integer);
  type
    bar := boolean;
  var
    f : foo;
    b : boolean;
  begin
    f := foo.new();
    f.a := 7;          // line 15
    f.a.print();
  end;

begin
  // ...
end;
```

Figure 23: Symbol searches cannot always begin at the local scope

that symbol was an alias for the integer class. The solution is to return the scope of the declaration along with the other information about the symbol a, and to start searching only at that scope for information about bar. So each object representing a type variable has a field to store this scope of declaration, and the method to retrieve symbol information searches accordingly.

6 Code Generation

Code generation is the second phase of the Leda compiler. It begins by sending the message genCode to the heterogeneous tree of objects—the syntax tree—created during the compiler's first phase. The top node of the tree is an object of the class programASTroot which is a subclass of the previously mentioned programASTnode, the class representing a Leda subprogram. By subclassing we can override the genCode method (recall Figure 22) to include operations unique to the main procedure of the source program, such as the creation of the constant pools discussed in Section 6.1. Following this, a genCode message is sent to the type section, and here class objects are laid out and type assignments are made as described in Section 6.2. Section 6.3 discusses code generation for the variables, expressions, and assignments found in the statements of a Leda subprogram. Finally, Section 6.4 addresses code generation for generic methods, and contrasts our direction with templates, another means of implementing parameterized types.

6.1 Integer constants

Integer, real, and enumerated constants are a bit more complicated to manage in **Leda** than in more conventional languages. This is due to the extension of the object model to even these basic types. Although the small experimental **Leda** programs written to date have performed satisfactorily, literal implementation of these semantics have potential to work against creating efficient object programs. It is expected that as the language definition is reworked and refined through experimentation, techniques will be employed which enhance the generated code while maintaining the semantics. Meanwhile, we will explain our system for generating code for integer constants. The explanation applies in concept to the real and enumerated constants as well.

During the first phase of the compiler, when the parser encounters an integer constant, an object of class **intExpNode** is created and placed in the syntax tree. Upon creation of the object, the class's constructor sends a message to the globally defined object **intValueRoot** which is the first node in a list of unique integers. In response to the message, this instance of class **intValueNode** responds by adding the new value if it does not already occur in the list. At code generation time, the root node of the syntax tree sends the **genCode** message to **intValueRoot** which generates code mapping out an object for each integer in the list. A naming convention is used consisting of the literal *IC_n* (for Integer Constant) where n is the integer being represented. When an **intExpNode** receives the **genCode** message, it composes a label corresponding to the value it contains, and generates the assembly language statement which moves the object into an address register. Another convention employed by the compiler is to use a particular address register ($a0$ in our case), where the evaluated result of an expression is to be stored. Thus when a message is sent to an expression to generate code, the sender may have no idea what sort of expression is generating code for itself, yet it can be sure that the generated code will leave the result in $a0$. Figures 24 and 25 show the various components discussed in this section.

6.2 Type Section

The **type** section consists of a series of assignment statements. The lvalue of one of these assignments must be a variable of type **metaclass** which we call a *type variable*. We do not require programmers to explicitly declare these variables in the **var** section, the compiler does that for them. The right hand side must be a valid **Leda** type—a class or function definition, an enumerated type, or type variable, which must be instantiated if it refers to a parameterized class. Class, function, and enumerated type definitions may be thought of as constants of type **metaclass**. These assignments are semantically equivalent to the standard assignment statements from the code section. In fact, run-time code is generated at the start of each subprogram to carry them out. None of this is to imply that classes are first class objects in **Leda**. They *are* objects as noted in Section 4; they can be used for member access, but type variables may not be assigned-to outside of the **type** section. That is, the class associated with a type variable cannot change during run-time as it might in a more flexible language like Smalltalk. The compiler depends on this fact and it would take a major revision to add the feature to **Leda**.

```
class intExpNode : public ASTnode {
private:
  int value;
  ASTnode *type;
public:
  intExpNode(int val)
  { value=val; type = intTypeObj;        // type integer
    if (intValueRoot) intValueRoot->addValue(val);
    else intValueRoot = new intValueNode(val);
  }
  ASTnode *genCode();
  // other methods
};


ASTnode *intExpNode::genCode()
{
  move(compLabel("IC_", itoa(value)), "a0");  // put the int object in reg a0
  return intTypeObj;
}
```

```
IC_28:                                  | the object '28'
                .long    IC_28_inst     | points to instance table
IC_28_inst:
                .word    1              | ref count
                .long    C1_shared      | shared table for class integer
                .long    28             | the value of '28'
```

Figure 24: An integer expression node, its code generator, and the perfect number 28 in memory

```
class valueNode {
protected:
  valueNode *next;
public:
  // virtual methods
};


class intValueNode : public valueNode{
private:
  int value;
public:
  intValueNode(int v) { value = v; next = NULL; }
  void addValue(int);
  ASTnode *genCode();
};


ASTnode *intValueNode::genCode()
{
  comment("---> intValueNode::genCode()");

  putLabel(compLabel("IC_", itoa(value)));          // lay out the object
  resLong(compLabel("IC_", itoa(value), "_inst"));
  putLabel(compLabel("IC_", itoa(value), "_inst"));
  resWord("1");
  resLong("C1_shared");
  resLong(itoa(value));

  if (next) next->genCode();
  return(NULL);
}
```

Figure 25: Maintaining a list of unique integer constants and laying them out in memory

```
void declTypeID::genCode(int count = 0)
{
  int offset, classNum;

  type->genCode();  // lays out class for type, and its type parameters

  // generate code to carry out the assignment
  classNum = classID->getClassStructure()->getClassNum();
  offset = -4 * (count + 1);
  move(compLabel("C", itoa(classNum)), regOffset("a4", offset));
  if (next) next->genCode(count + 1);
}
```

Figure 26: Generating code for a type declaration

Figure 26 shows the method which implements the **genCode** message sent to a subprogram's list of type declarations. A type declaration responds to the message by sending the same message to the type (right hand side) of the declaration. Figure 27 shows the implementation of that message for a user-defined class. The method lays out the class object and its shared table, and passes the message on to any type parameters the class might have. Predefined classes such as **integer** ignore the **genCode** message since their class objects are already laid out in the special assembly module. Type variables also ignore the message, since they are merely aliases for other classes which will take care of themselves when they receive the message personally. In any case, control is returned to the type declaration object, which next generates code to carry out the assignment, and finally sends the **genCode** message to the next type declaration in the list. The method exhibits an unfortunate degree of coupling as it depends on the fact that the type variables are added to the list of variable declarations before any other variables, and in the same order that they occur in the **type** section. A count is maintained by the **genCode** method so that the type declarations know where in the activation record to effect the assignment.

6.3 Statements

To generate code for the statements within a subprogram, the subprogram node sends the **genCode** message to the list of statements. Each statement generates the appropriate code for itself— usually by sending the **genCode** message to its component expressions—and then sends the message on to the next statement in the list. Much of this work is accomplished using straightforward code-generation techniques. The following subsections describe various types of statements and expressions and their code generation, particularly those aspects unique to the **Leda** compiler.

6.3.1 Variables and Member Access

A variable in **Leda** is recursively defined to be an identifier which may be followed by the membership operator and another variable. Each variable in the chain must be a member of the declared class,

```
ASTnode *userclassType::genCode()
{
  data_seg();
  put(str("C", itoa(uniqueScopeNum)));               // label the class
  rlong(str("C", itoa(uniqueScopeNum), "_inst")); // Point to inst table
  put(str("C", itoa(uniqueScopeNum), "_inst"));   // Begin inst table
  rword("1");                                       // Ref count
  rlong("C0_shared");                               // Ptr to shared table
  rlong(str("C", itoa(uniqueScopeNum), "_constr_code"));  // Ptr to constructor
  rlong("0");    // constructor's environment - 0 for now
  rlong(itoa(6 + 4 * ((offsets) ? offsets->countInst() : 0)));  // size of instance
  rlong(str("C", itoa(uniqueScopeNum), "_shared"));             // location of shared table

  if (sharedVars) sharedVars->genSharedVar(uniqueScopeNum);     // rest of instance vars

  put(str("C", itoa(uniqueScopeNum), "_shared")); // begin class's shared table
  rlong(str("C", itoa(uniqueScopeNum), "_inst")); // Ptr back to class

  if (superClass) {                                 // Ptr to superclass
    rlong(str("C", itoa(scope::deepest
      ->getClassDef(superClass->getName())->getClassNum()), "_inst")); }
  else
    rlong("0");

  offsets->genSharedTable();                        // Rest of shared table

  text_seg();       // lay out constructor
  bra(str("C", itoa(uniqueScopeNum), "_constr_end"));
  put(str("C", itoa(uniqueScopeNum), "_constr_code"));
  link(reg("a6"), 0);
  push(str("C", itoa(uniqueScopeNum)));      // new expects class on stack
  pushInt(0, "static link for new");
  jsr("C0_new_code");
  offsets->genConstructorAssignments();      // gen code for assignments

  move(regOff(reg("a6"),4), reg("a1"));      // callee pops args off
  loadea(regOff(reg("a6"), 12 + 4*offsets->countInst()), reg("sp"));
  move(regOff(reg("a6"),0), reg("a6"), "restore frame ptr");
  jump(regOff(reg("a1"),0));
  put(str("C", itoa(uniqueScopeNum), "_constr_end"));

  // lay out the class for any type parameters
  if (paramTypes) paramTypes->genCode();
}
```

Figure 27: Generating code for a user-defined class

or *static type*, of the variable which precedes it. Variables are represented in the syntax tree by objects of class IDnode which include the name of the identifier and a pointer to the next variable. Generating code for a variable in a Leda source program consists of putting the object or lvalue referred to by the variable into register $a0$, as per the convention mentioned above. Normally, the object that requests a variable to generate code expects the object itself to be placed in $a0$, but in some cases, the left-hand side of an assignment for example, require the address of the object to be generated instead. For this reason, the genCode method for a variable contains a boolean argument which signals the method to either dereference the variable or not. In either case, the variable sends a message to the local scope to generate an address, and then adds a step for dereferencing if required.

Two pieces of information are necessary to achieve this—the scope of definition of the variable, and its relative position within the activation record associated with that scope. When a variable receives a message to generate code for itself it sends a genCodeID message to the current local scope to generate the code, which takes the name of the variable as an argument. If the scope is a subprogram (as opposed to a class), it sends the genCodeID message to its argument, constant, and variable declarations to generate the code. If none of these lists find the name for which code is to be generated, the scope passes the message up to the next higher level. When a class receives the genCodeID message, it checks its instance and shared variables for the name. If found, the class generates code for the receiver, which it knows is in the position of the first parameter within the local scope. It then generates code for the variable as if it were coded as a class member accessed via the reserved word self. The difference between the current scope and the level in which the variable declaration is found is maintained so that code can be generated to travel down the static chain the right number of links to reach the appropriate activation record. The genCodeID methods for subprogram nodes and variable declarations are shown in Figure 28.

When a variable of the form *a.b.c* is encountered, code for the variable *a* is generated as described above. Before exiting, the method checks to see if there is some member being accessed, which in this case there is, namely *b*. The variable then sends itself the message genCodeMember to generate code for *b*. The reason that this message is sent to *a* and not *b* is that the code has to generated in the context of the class of *a*, something which *b* doesn't know anything about. The methods for genCode and genCodeID for the class IDnode are shown in Figure 29. To generate code for a class member, the method gets the necessary information from the offset table associated with the class of the variable through which the member is being accessed. One of two different messages is sent, offset or metaOffset, depending on whether the offset table should think of itself as representing a class or a metaclass. The information obtained is sent to the non-method function genMember shown in Figure 30. This function generates the code that puts the member variable (or its address, depending again on a boolean argument) in register $a0$. For shared members the generated code checks to see if the variable is undefined in which case there is no shared table pointer through which to access the member. In this situation, the shared table associated with the variable's static type is used.

6.3.2 Assignments

Assignment statements respond to the genCode message by having the left-hand side generate code for itself without dereferencing. That value, the address of the object being assigned to, is then put

```
// The genCodeID method for a subprogram
ASTnode *programASTnode::genCodeID(char *name, int level_diff, invInd invk)
{ ASTnode *lt;

  // look for ID in constants, locals, then parameters. If not found,
  //    increase level difference and try again in parent level unless
  //    already at top in which case return NULL.

  if (receiver && !strcmp(name, "self"))       // special case for "self"
    return receiver->genCodeID("self", 0, 0);
  else {
    if (!constantdefs || !(lt = constantdefs->genCodeID(name)))
      if (!variabledefs || !(lt = variabledefs->genCodeID(name,level_diff, 0)))
        if (!typedefs || !(lt= typedefs->genCodeEnumID(name)))
          // relative position of args depends on if there is a receiver
          if (!receiver)
            lt = args ? args->genCodeID(name, level_diff, 0) : NULL;
          else  lt = args ? args->genCodeID(name, level_diff, 1) : NULL;

    if (!lt && parentLevel)
      return parentLevel->genCodeID(name, ++level_diff, invk);

    return lt;
  }
}

// The genCodeID method for a variable declaration
ASTnode *declVarID::genCodeID(char *IDname, int level_diff, int location)
{ int k;

  // if name matches this declaration, put address of variable in a0
  if (!(strcmp(name, IDname))) {
    move("a4","a0", "move frame pointer to a0");
    comment("follow static chain back for each level");
    if (level_diff >= 1)
      for (k=1; k <= level_diff; k++)
        move(regOffset("a0", 8), "a0");
    sub((location+1)*4, "a0");
    return type;
  }
  // otherwise, increase relative location and try next declaration
  else if (next)
     return next->genCodeID(IDname, level_diff, ++location);
  else return NULL;
}
```

Figure 28: Generating code for a variable

```
ASTnode *IDnode::genCode(bool deref)
{
  ASTnode  *type;

  type = scope::deepest->genCodeID(name, 0, NOINVOKE);
  if (deref) move(regOffset("a0", 0), "a0");

  if (next)  // is a class member being accessed via this variable?
    return this->genCodeMember(type, deref);
  else
    return type;
}


ASTnode *IDnode::genCodeMember(ASTnode *varType, bool deref)
{
  int          offset;
  memberType   instORshrd;
  ASTnode      *varTypeNext;

  // Should the class see itself as a metaclass in this case?
  if (varType->getLedaType() == METACLASS)
    varTypeNext = this->metaOffset(next->name, &offset, &instORshrd);
  else
    varTypeNext = varType->offset(next->name, &offset, &instORshrd);

  genMember(instORshrd, offset, varType, deref);

  if (next->next)  // is there yet another class member being accessed?
    return next->genCodeMember(varTypeNext, deref);
  else
    return varTypeNext;
}
```

Figure 29: Generating code for a variable and its members

```
void genMember(char instORshrd, int offset, ASTnode *objType, bool deref=TRUE)
{ int label;

  // for instance variables pull object out of instance table, which reg a0
  //   happens to be pointing to:
  if (instORshrd == INST)
    if (deref)
      move(regOffset("a0", offset), "a0");
    else
    {
      move(regOffset("a0", 0), "a0");
      add(offset, "a0");
    }
  else
    // For shared variables, use pointer to shared table, or shared table
    //   associated with class of definition if variable is undefined.
    if (deref)
    {
      makeLabelsL(label, 2);
      compare("0", "a0");        // if undefined, use class of definition
      bne(compLabel("L", itoa(label)));
      loadea(compLabel("C", itoa(objType->getClassID()->getClassStructure()
                               ->getClassNum()), "_shared"), "a0");
      braLabel(compLabel("L", itoa(label+1)));
      putLabel(compLabel("L", itoa(label)));
      move(regOffset("a0", 2),"a0", "put address of shared table in a0");
      putLabel(compLabel("L", itoa(label + 1)));
      move(regOffset("a0", offset), "a0", "ptr to object in a0");
      move(regOffset("a0", 0), "a0", "put object in a0");
    }
    else
    {
      makeLabelsL(label, 2);
      compare("0", regOffset("a0", 0));
      bne(compLabel("L", itoa(label)));
      loadea(compLabel("C", itoa(objType->getClassID()->getClassStructure()
                               ->getClassNum()), "_shared"), "a0");
      braLabel(compLabel("L", itoa(label+1)));
      putLabel(compLabel("L", itoa(label)));
      move(regOffset("a0", 0), "a0");
      move(regOffset("a0", 2), "a0");
      putLabel(compLabel("L", itoa(label + 1)));
      move(regOffset("a0", offset), "a0");
    }
}
```

Figure 30: The genMember function

on the stack for safe keeping, and a message is sent to the expression on the right-hand side for it to generate code leaving its value in register $a0$. The variable of the left-hand side is made to refer to this new value and the assignment is complete.

Although garbage collection is not fully implemented, the reference count is being maintained by the code generated by the assignment statement. The reference count of the old object referred to by the lvalue of the assignment is decremented; that of the new one is incremented. Care must be taken to check that these objects are defined before erroneously attempting to follow a null pointer toward an illusive reference count. The genCode method for the assignment statement is shown in Figure 31.

6.3.3 Binary expressions and operator overloading

Binary operators in Leda are treated as messages sent to the object which results from evaluating the left-hand expression. The value of the right-hand expression is an argument to the method. For example, the expression x * 7 is interpreted as sending the message times to the receiver x with the argument 7. In fact, the expression x.times(7) is wholly equivalent. Every binary operator has a corresponding name, some of which are shown in the following table.

Binary operators:	Name
+	plus
−	minus
*	times
/	slash
%	mod
=	equal
<>	notEqual
>	greater
<	less
>=	greaterEqual
<=	lessEqual
&	and
|	or

When the compiler encounters a binary expression it generates code for the right-hand side and pushes it on the stack as an argument. It then generates code for the left-hand expression and pushes it on the stack as the receiver. Next it finds the offset for the method with the name corresponding to the operator within either the instance or shared tables of the class of the object returned by the left-hand side, and generates code which places the method-object in register $a0$. Finally code is generated to invoke the method. Figure 32 shows the method in the class representing a binary expression which generates this code.

By implementing these semantics of binary expressions we have given the Leda programmer the ability to overload any of the binary operators for use with any programmer-defined classes. This is done by declaring a method with one argument (other than the receiver), which uses the name associated with the operator to be overloaded. Figure 33 shows how the plus sign can be overloaded

```
ASTnode *assignStatem::genCode()
{
  ASTnode *lvalType, *rvalType;

  // put the lvalue in a0, and save it on the stack
  lvalType = variable->genCode(NO_DEREF);
  push("a0", "push address on stack");

  // evaluate right hand side and put resulting obj in a0
  rvalType = expression->genCode();

  // carry out the assignment and adjust the reference counts
  incrRefCnt("a0");
  pop("a1");
  move(regOffset("a1", 0), "a5");
  decrRefCnt("a5");
  move("a0", regOffset("a1", 0));

  return(NULL);
}

// increment/decrement reference count for object in address register *An
void incrRefCnt(char *An)
{
  int label;

  makeLabelsL(label,1);
  compare("0", An);  // don't bother if object is undefined
  beq(compLabel("L", itoa(label)));
  addquiw(1, regOffset(An, 0));
  putLabel(compLabel("L", itoa(label)));
}

void decrRefCnt(char *An)
{
  int label;

  makeLabelsL(label,1);
  compare("0", An);  // don't bother if object is undefined
  beq(compLabel("L", itoa(label)));
  subquiw(1, regOffset(An, 0));
  putLabel(compLabel("L", itoa(label)));

  //  code here to deallocate memory if refcnt hits 0
}
```

Figure 31: Generating code for an assignment statement

```
ASTnode *binaryExpNode::genCode()
{ int        offset;      // offset of the method within...
  memberType instORshrd; //    ...shared or instance table
  ASTnode    *lcType;     // Where to look for the method
  ASTnode    *opType;     // Return type of the method

  rightChild->genCode();     // eval right child and push on stack
  push("a0");
  incrRefCnt("a0");

  lcType = leftChild->genCode();  // eval receiver and push on stack
  push("a0");
  incrRefCnt("a0");

  // search for the operation name in the class table of
  //    the receiver type, error if not found:
  if (lcType->getLedaType() == METACLASS)
    opType = leftChild->metaOffset(op, &offset, &instORshrd);
  else
    opType = lcType->offset(op, &offset, &instORshrd);

  // put the method-object in a0
  genMember(instORshrd, offset, lcType, DEREF);
  genJsr();  // generate code to jump-subroutine to the code (shown below)
  pop("a1", "pop receiver off stack");
  decrRefCnt("a1");
  pop("a1", "pop parameter off stack");
  decrRefCnt("a1");
  return opType->getReturnType();
}

void genJsr()
{ escInd cInd;
  move(regOffset("a0", 6), "a1", "put pointer to code in a0");
  push(regOffset("a0", 10), "push static link on stack");
  jsr(regOffset("a1", 0));
  comment("pop static link off stack");
  addquil(4, "sp");

  // If there is an escaping closure, must treat differently
  if ((cInd = scope::deepest->getClosureInd()) == ESC) {
    move(regOffset("a6", 8), "a4", "restore a4"); // gets ruined in proc call
  } else
    move("a6", "a4");
}
```

Figure 32: Generating code for a binary expression

```
type
  class := Point
      x : integer;
      y : integer;
    shared
      plus : method(Point)->Point;
      // other methods
  end;

var
  p1, p2, p3 : Point;

  method Point.plus(p : Point)->Point;
  begin
    return Point(x + p.x, y + p.y);
  end;

begin
  p1 := Point(3,7);
  p2 := Point(5,11);

  p3 := p1 + p2;    // equivalent to p3 := p1.plus(p2)
                    // p3 is now the point (8, 18)
end;
```

Figure 33: Overloading the binary operator + in class Point

for use with objects of class Point.

6.4 Generic Methods

Generating a single block of code for a generic method may at first seem difficult. Some entities within the method are not bound to concrete types at compile time. We anticipated this problem in two ways. First, the uniform object representation assures that all values take up the same amount of space (the size of a pointer) inside the activation record or offset table in which they reside. Thus the compiler need never be concerned with types of variables in order to "find" them at run-time. Second, we put restrictions on the use of variables declared to be an instance of a type parameter. We never put the compiler in a position of needing any more information with regard to how an object should respond to a message beyond that which may be guaranteed by a constraint on the type parameter. When we add to this structure the fact that type parameters are treated as locally defined types from within their class (which includes the member methods), one finds that no special code-generation techniques are necessary. Truly polymorphic code, identical statements correctly and safely computing over multiple types, is generated with no special attention from the compiler. In choosing this direction we have gained simpler code generation, smaller object programs, and what we perceive as semantic elegance; but there is a cost. Increased complexity of type checking, and less flexibility for the programmer must be considered. The tradeoffs are not well understood. The area of parameterized-type research is relatively young, especially with regard to practical experience.

Most implementations of parameterized types can be put into one of two categories. Some, like Leda, arrange things so that a single block of polymorphic code can be generated without concern as to how the class will eventually be instantiated. The languae Eiffel takes this approach [Mey88]. Eiffel goes even further than Leda in that it restrics programmers additionally by not allowing them to constrain type parameters. Eiffel gains by simplifying the implementation still more. The other category generally lifts restrictions on the programmer by providing parameterized types in the form of *templates*. An implementation of this sort is described in [E&S90] as a proposal for adding the facility to C++. Templates can be thought of more as a macro. In Leda instantiation is a passive act in which a programmer makes use of an implicitly created class which shares singular generic methods with other members of its sub-hierarchy. Instantiating a template is an active, constructive act, creating a new class where none had existed. The new class's connection to other classes instantiated from the same template is superficial—a common look, a structure. Methods declared within a class template must eventually exist in multiplicity to serve the various classes instantiated from the template. The other side of the tradeoff is stated eloquently in [E&S90]:

> Speecifying no restrictions on what types can match a type argument gives the programmer the maximum flexibility. The cost is that errors—such as attempting to sort objects of a type that does not have comparison operators—will not in general be detected until link time. Only then are both the template and the type of elements to be sorted available.

7 Conclusions

We have written and implemented a compiler for Leda, a new language designed with the goal of furthering research in the area of multi-paradigm languages and algorithms. Our version of the compiler was written to enable the first practical experimentation with the language, expected to result in an evolution of both the language and its compiler. As such we concentrated on refining Leda's syntax and semantics, implementing them more or less directly without regard to a high level of optimization. We feel we have succeeded with a useable compiler which has already served to generate ideas about the future of the language.

We saw as our challenge the creation of a language which is simple and elegant, while providing the ability to solve problems utilizing various points of view or *paradigms*. In addition to some of the more standard features found in popular modern high-level languages, parameterized classes are included to enable a high-level of abstraction without sacrificing the safety of strong typing. Continued research will determine the usefulness of this feature, particularly how it may serve as a vehicle for combining the different paradigms. Type parameters are given a dual semantics. From inside a class they act as a locally defined type whose scope is the class itself and its member subprograms. From outside the class they serve as a shorthand for introducing a set of classes—a sub-hierarchy—any of which the programmer may utilize through instantiation.

The desire to generate truly generic code for member subprograms of parameterized classes pointed to the need for a uniform representation of all entities regardless of their type. It was decided that everything would be an object and all types classes. As much as the decision smoothed the implementation, it caused concern for the efficiency of Leda object programs. We also noted the competing notion of templates as a different means of implementing parameterized types. Our hope is that Leda will help us gain insight into the tradeoffs involved with these differing approaches.

Although we are satisfied with the performance of small programs we have written, more and larger programs need to be tested. Inspired by the experience of the Smalltalk implementors, we suspect that more sophisticated implementation techniques, if necessary, will keep the present semantics viable.

Symbol information is collected for type checking and code generation. The information is kept within the syntax tree created during the first phase of compilation from the Leda source program. A pivotal structure used to maintain type information is the classID. This structure makes for a clean and maintainable implementation, but no doubt more efficient techniques could be used. Some, including a flat string representation combined with a hashing scheme, are being considered.

The Leda compiler is written in an object-oriented style. Code generation is sparked by a message to the root of the syntax tree. The source-program components that make up the nodes of the tree take responsibility for generating code for themselves, and sending the message on to their constituents. Presently, some work done while collecting symbol information is being duplicated during code generation. This allowed concurrent developement of, and experimentation with, the two compiler phases by different implementors. Future versions of the compiler should eliminate this redundancy.

Some work remains for Leda to become the general purpose programming language to which it aspires. Presently, output is rudimentary, serving mainly to test the currently implemented features of the language. The problem of input/output needs to be solved. It also must be decided what other built-in classes should be provided, such as strings, arrays, streams, and perhaps others. Along with these come the question of how much should be implemented with primitive data types, unavailable to the Leda programmer, versus what should be implemented using the language itself. Large programs using significant memory resources will require the garbage collection system to be fully implemented. The code-generation phase needs to be augmented to output information enabling source-level debugging. Research already underway aims to make Leda part of a comprehensive multi-paradigm programming environment [Pan91].

A The Compiler

Besides knowledge and experience, the efforts described by this paper led to a working Leda compiler. It is currently able to generate running code for two different computers, both based on Motorola 68000 series processors. Porting the compiler to other machines based on the same style processor is straightforward, and should only involve changes to a single module to adjust for the particular assembly-language dialect. Different architectures will complicate the porting process.

A language definition corresponding to this version of the compiler is given in [S&P91]. The rest of this appendix gives some statistics about the compiler and the programs it generates.

(a)
```
#include <stdio.h>
#define N 20

int nthFib(int n)
{
  if (n <= 2) return 1;
  return nthFib(n-2) + nthFib(n-1);
}

main()
{
  int i, fib;
  for (i=0; i<20; i++)
    fib = nthFib(N);
  printf("%d\n", fib);
}
```

(b)
```
const
  N := 20
var
  i, fib : integer;
  nthFib : function(integer)->integer;

begin
  nthFib := function(n : integer)->integer;
            begin
              if n <= 2 then return 1;
              return nthFib(n - 2) + nthFib(n - 1);
            end;

  for i := 1 to 20
     fib := nthFib(N);
  fib.print();
end;
```

Figure 34: Doubly recursive functions to compute the 20th fibinacci number in (a) C and (b) Leda

Target Machines:

1. HP 9000/375; HP-UX Operating System
2. Tektronix Tek4315; UTek Operating System

Compiler Size:
Source: App. 7600 lines in 9 seperate files
Executable: App. 750K bytes

The following table pertains to the two programs shown in Figure 34. One is written in Leda, the other in C. The C program was compiled under the GNU gcc compiler. Each computes the Nth fibinacci number for some constant N. The computation is repeated 20 times (for more meaningful timings) before the result is printed. The algorithm is purposely naive, using double recursion to test the compilers' function calling facilities and stack limits. Times are given in seconds. Object size is in bytes.

Fibinacci	Leda	C
Compilation Time	9	11
Object Size	28672	16876
Running Times:		
N = 15	3	.13
N = 20	41	1.4
N = 22	122	4
N = 25	<stack full>	16

The next table refers to two programs which find all 92 solutions to the 8-queens problem. The approach is object-oriented, adapted from a Smalltalk program given and explained in [Bud87]. For contrast we wrote the program in C++ as well, compiling it under the GNU g++ compiler. The text of each program is given in a subsection below.

8 Queens	Leda	C++
Compilation Time	17	28
Object Size	34558	32768
Running Time	17	1

A.1 8 queens in Leda

```
type
  Queen := class
      row : integer;
      column : integer;
      neighbor : Queen;
    shared
      print : method();
      first : method();
      next  : method()->Queen;
      testPosition : method()->Queen;
      checkRowCol  : method(integer, integer)->boolean;
  end;

var
  lastQueen : Queen;
  i : integer;

method Queen.print();
begin
  if defined(neighbor) then neighbor.print();
  column.print(); row.print();
end;

method Queen.first();
begin
  if defined(neighbor) then
    neighbor.first();
  row := 1;
  self.testPosition();
end;

method Queen.next()->Queen;
var
  nilQueen : Queen;
begin
  if row = 8 then
    if defined(neighbor) & defined(neighbor.next()) then
      row := 0
    else
      return nilQueen;
  row := row + 1;
  return self.testPosition();
end;

method Queen.testPosition()->Queen;
begin
  if defined(neighbor) then
    if neighbor.checkRowCol(row, column) then
      return self.next();
  return self;
end;
```

```
method Queen.checkRowCol(testRow, testCol : integer)->boolean;
var
  columnDifference : integer;
begin
  columnDifference := testCol - column;
  if   (row = testRow)
    |  ((row + columnDifference) = testRow)
    |  ((row - columnDifference) = testRow)
  then
    return true;

  if defined(neighbor) then
    return neighbor.checkRowCol(testRow, testCol)
  else
    return false;
end;

// M A I N //
begin
  // initialize queens
  for i := 1 to 8
    lastQueen := Queen(NIL, i, lastQueen);

  // first solution
  lastQueen.first();
  lastQueen.print();

  i := 1;

  while defined(lastQueen.next())
    i := i + 1;
  i.print();
end;
```

A.2 8 Queens in C++

```
#include <stream.h>
#define FALSE 0
#define TRUE  1

typedef int BOOL;

class Queen {
public:
    int row;
    int column;
    Queen* neighbor;

  // methods
    virtual void print();
    virtual void first();
    virtual Queen *next();
    virtual Queen *testPosition();
    virtual BOOL checkRowCol(int, int);
    Queen(int r, int c, Queen *q)
      { row=r; column=c; neighbor=q; }
};

void Queen::print()
{
  if (neighbor) neighbor->print();
  printf("%d, %d\n", column, row);
}

void Queen::first()
{
  if (neighbor)
    neighbor->first();
  row = 1;
  this->testPosition();
}

Queen *Queen::next()
{
  if (row == 8)
    if (neighbor && neighbor->next())
      row = 0;
    else
      return NULL;
  ++row;
  return this->testPosition();
}

Queen *Queen::testPosition()
{
  if (neighbor)
    if (neighbor->checkRowCol(row, column))
      return this->next();
  return this;
}

BOOL Queen::checkRowCol(int testRow, int testCol)
{
  int columnDifference;

  columnDifference = testCol - column;
  if   ((row == testRow)
     || ((row + columnDifference) == testRow)
     || ((row - columnDifference) == testRow))
  // then:
       return TRUE;

  if (neighbor)
    return neighbor->checkRowCol(testRow, testCol);
  else
    return FALSE;
}


main()
{
  Queen *lastQueen = NULL;
  int i, j;

  lastQueen = NULL;
  // initialize queens
  for (i=1; i<=8; i++)
    lastQueen = new Queen(0, i, lastQueen);

  // first solution
  lastQueen->first();
  lastQueen->print();

  i = 1;

  while (lastQueen->next())
    i = i + 1;
  printf("%d\n", i);
}
```

References

[B&D73] Birtwistle, G. and O. Dahl and B. Myrhaug and K. Nygaard, *Simula Begin*, Auerbach Pub., New York, NY (1973)

[Bob84] Bobrow, D. G., "If Prolog is the Answer, What is the Question?" *Proc. International Conference on Fifth Generation Computer Systems 1984*

[Bud87] Budd, T., *A Little Smalltalk*, Addison-Wesley, Menlo Park, CA (1987)

[Bud89a] Budd, T. A., "Leda: A Blending of Imperative and Relational Programming," *IEEE Software*, January 1991

[Bud89b] Budd, T. A., "Low Cost First Class Functions," Oregon State University, Technical Report 89-60-12, June 1989. *submitted for publication.*

[Bud89c] Budd, T. A., "Data Structures in Leda," Oregon State University, Technical Report 89-60-17, August 1989.

[Bud91a] Budd, T. A., "Sharing and First Class Functions in Object-Oriented Languages," Working document, Oregon State University, February 1991]

[Bud91b] Budd, T. A., "Avoiding Backtracking by Capturing the Future," Working Document, Oregon State University, 1991

[Bud91c] Budd, T. A., "Multiparadigm Data Structures in Leda," Working document, Oregon State University, April 1991

[Che91] Cherian, V., "Implementation of First Class Functions and Type Checking for a Multiparadigm Language," *Research Paper for M.S. Degree*, Oregon State University, May 1991

[Dij72] Dijkstra, E. W., "The Humble Programmer," *Communications of the ACM*, October 1972

[Dij76] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ (1976)

[E&G84] Einarsson, B. and W. M. Gentleman, "Mixed Language Programming," *Software Practice and Experience*, April 1984

[E&S90] Ellis, M. and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Menlo Park, CA (1990)

[G&R89] Goldberg, A., and D. Robson., *Smalltalk-80: The Language*, Addison-Wesley, Menlo Park, CA (1989)

[Hai86a] Hailpern, B., "Multiparadigm Languages and Environments," *IEEE Software*, January 1986

[Hai86b] Hailpern, B., "Multiparadigm Research: A Survey of Nine Projects," *IEEE Software*, January 1986

[H&S90] Hayes, R. and N. C. Hutchinson and R. D. Schlichting, "Integrating Emerald into a System for Mixed-Language Programming," *Computer Languages*, Vol.15 No.2 (1990)

[J&G86] Jenkins, M. A. and J. I. Glasgow and C. D. McCrosky, "Programming Styles in Nial," *IEEE Software*, July 1986

[K&E88] Koschmann, T. and M. W. Evens, "Bridging the Gap between Object-Oriented and Logic Programming," *IEEE Software*, July 1988

[Lan91] "The Language List—version 1.4 9/8/91: Information on Aprroximately 1300 computer languages, past and present." Maintained by Bill Kinnersley, Computer Science Department, University of Kansas (billk@hawk.cs.ukans.edu)

[Mey88] Meyer, B., "Object-Oriented Software Construction," Prentice Hall, New York, NY (1988)

[Pan91] Pandey, Rajeev K., "Sparta: A Programming Environment for the Multiparadigm Language Leda," *Research Proposal*, Oregon State University, Work in Progress

[Pes91] Pesch, W., "Implementing Logic in Leda," Oregon State University, Technical Report 91-60-10, September 1991

[Pla91] Placer, J., "Multiparadigm Research: A New Direction in Language Design," *Sigplan Notices*, March 1991

[S&P91] Shur, J. and W. Pesch, "A Leda Language Definition," Oregon State University, Technical Report 91-60-9, August 1991

[Ste90] Steele, G. L, *Common Lisp—The Language (second edition)*, Digital Press, Bedford, MA (1990)

[S&B86] Stefik, M. J. and D. B. Bobrow, and K. M. Kahn, "Integrating Access-Oriented Programming into a Multiparadigm Environment," *IEEE Software*, January 1986

[Weg76] Wegner, P., "Programming Languages, the First 25 Years (1976)," *Programming Languages: A Grand Tour*, edited by Ellis Horowitz, Computer Science Press, Rockville, MD (1987)

[Wol88] Wolfram, S. *Mathematica*, Addison-Wesley, Menlo Park, CA (1988)

[Wu91] Wu, S., "Integrating Logic and Object-Oriented Programming," *Oops Messenger*, January 1991