OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

Implementing Logic in **Leda**

Wolfgang Pesch
Department of Computer Science
Oregon State University
Corvallis, OR 97331

91-60-10

Implementing Logic in **Leda**

Wolfgang Pesch
Department of Computer Science
Oregon State University
Corvallis, OR
97331
peschw@mist.cs.orst.edu

September 30, 1991

Abstract

Leda is a newly evolving, strongly typed, compiled multi-paradigm programming language. This paper describes the integration of one of its supported paradigms, the logical (or relational) paradigm, into the language and the current implementation. It also describes implementational aspects of enumerated types and gives a variety of example programs that demonstrate the usefulness of the achieved blending between the logical and other existing programming paradigms in **Leda**.

1 Introduction

Leda is a strongly typed, compiled multi-paradigm language, which is currently under development at Oregon State University. A paradigm, by its very definition, offers a single-minded, cohesive view, which helps one to think clearly about a problem. A programming paradigm, which can be thought of as an abstraction based on features of existing programming languages, allows a programmer to use a restricted set of concepts. Each programming paradigm embodies a very specific approach to computation and thus affects the way one thinks about problem solving and algorithms in general. Hence the major goal in the research area of multi-paradigm programming languages is to investigate ways in which multiple paradigms can be beneficial to each other if they are made available to the programmer within a single linguistic framework. **Leda** was designed to be a vehicle for this undertaking, and it encompasses the imperative, functional, object-oriented and logical programming paradigms.

Work on the implementation of the **Leda** compiler began as a research group project in January 1990 and concluded in the fall of 1991. A general overview of the language as well as its raison d'être may be

found in [Bud89b] and [Bud89c], along with a language reference manual [ShP91], which describes the language as currently implemented. [Bud91c] envisions the use of multi-paradigm data structures. The functional paradigm and first class functions are covered in [Bud89a], [Bud89d], [Bud91b] and their actual implementation is found in [Che91]. The object-oriented paradigm and parameterized types are addressed in [Bud89d] and [Bud91b]. Their implementation as well as the overall structure and design of the **Leda** compiler are found in [Shu91].

This paper concentrates on the linguistic and implementational aspects of integrating the relational paradigm into the language. Original ideas of these aspects appear in [Bud91a]. Before getting to relational programming, section 2 is devoted to the implementation of enumerated types in the **Leda** compiler. This data structure had to be made available before serious work on relations could begin. Section 3 discusses syntactic and semantic aspects of embedding logical programming into the language and introduces the features present in the current system. Section 4 describes the implementation technique of these features along with necessary changes to other parts of the compiler. Section 5 presents a number of **Leda** programs, that demonstrate how the different paradigms (including the relational one) benefit each other.

2 Enumerated Types

This section describes the implementation of enumerated types in the **Leda** compiler. Its only connection with the main aspect of this paper (relational programming) is the fact, that enumerated types are most useful when used in relations. For the basic structure and design of the **Leda** compiler readers may be referred to [Shu91] and [ShP91].

A central aspect in the implementation of **Leda** is the fact that all entities are objects, which are instances of some class. Enumerated types are no exception. Furthermore, every data type in **Leda** is internally represented as an object of type CLASS. This class is not available to the programmer, but acts as a metaclass for defining new type objects [Shu91]. Because every **Leda** type is represented by a class object, the terms *class* and *type* will be used interchangeably throughout this paper.

2.1 User Defined Enumerated Types

An enumerated type in **Leda** is defined in the type section by assigning a parenthesized list of identifiers to an identifier, which is the name of the type. The list of identifiers is a set of enumerated constant values which variables declared to be of this type may take on. No enumerated constant may be declared twice, i.e. all enumerated constants must be unique. Figure 2.1.1 shows the definition of an enumerated type in **Leda** (for syntax and semantics of the language see [ShP91]):

```
type
  names:=(helen, leda, zeus);
```

Figure 2.1.1: Type declaration of an enumerated type

The value of an enumerated type is an object of type CLASS and is internally defined to be a subclass of the predefined abstract class **enum**. This abstract class includes the basic operations on enumerated types. It is not available to the programmer, but is defined in an assembly module which has to be linked with **Leda** programs as part of the compilation process. All user defined enumerated types inherit their operations from this class. Figure 2.1.2 shows the class definition of the abstract class **enum** in pseudo **Leda** syntax.

```
enum := class
  index : primitive;
shared
  numEntries : primitive;
  entryTable : primitive;
  print : method();
  succ : method()->enum;
  pred : method()->enum;
  less : method(enum)->boolean;
  lessEqual : method(enum)->boolean;
  greater : method(enum)->boolean;
  greaterEqual : method(enum)->boolean;
  equal : method(enum)->boolean;
  notEqual : method(enum)->boolean;
end;
```

Figure 2.1.2: Pseudo **Leda** syntax defining the class **enum**

```
C3:             .long   C3_inst       | pointer to instance table of
                                      | of class C3
C3_inst:        .word   1             | reference count for class object
                .long   C0_shared     | pointer to class CLASS
                .long   0             | not used
                .long   0             | not used
                .long   10            | size of an object of type enum
                .long   C3_shared     | pointer to shared table

C3_numEntries:  .long   0             | number of enumerated constants
C3_entryTable:  .long   0             | table of literal strings

C3_print:       .long   C3_print_inst       | repertoire of operations
C3_succ:        .long   C3_succ_inst        | on enumerated types
C3_pred:        .long   C3_pred_inst
C3_less:        .long   C1_less_inst
C3_lessEqual:   .long   C1_lessEqual_inst
C3_greater:     .long   C1_greater_inst
C3_greaterEqual:.long   C1_greaterEqual_inst
C3_equal:       .long   C1_equal_inst
C3_notEqual:    .long   C1_notEqual_inst
                                            | the shared table of C3:
C3_shared:      .long   C3_inst             | pointer to class itself
                .long   0                   | pointer to superclass
                .long   C3_print            | first shared variable
                .long   C3_succ
                .long   C3_pred
                .long   C3_less
                .long   C3_lessEqual
                .long   C3_greater
                .long   C3_greaterEqual
                .long   C3_equal
                .long   C3_notEqual         | last shared variable
```

Figure 2.1.3: Representation of class **enum** in memory

```
C3_succ_code:
                link    a6,#-0          | no local variables
                movl    a6@(12),a1      | put object in a1
                movl    a1@(6),d5       | put index in d5
                movl    a1@(2),a1       | a1 points to class table
                movl    a1@(8),a0       | a0 points to number of entries
                movl    a0@(0),d4       | d4 contains number of entries
                addl    #1,d5           | d5 contains successor index
                cmpl    d4,d5           | if index = num entries
                bne     enum_ret        | make it zero
                clrl    d5              | else return
                bra     enum_ret        | defined in C3_pred_code
C3_pred_code:
                link    a6,#-0
                movl    a6@(12),a1      | put object in a1
                movl    a1@(6),d5       | put index in d5
                movl    a1@(2),a1       | a1 points to class table
                movl    a1@(8),a0       | a0 points to number of entries
                movl    a0@(0),d4       | d4 contains number of entries
                subl    #1,d5           | d5 contains successor index
                bpl     enum_ret        | if index is negative
                movl    d4,d5           | make it (num entries - 1)
                subl    #1,d5
enum_ret:
                movl    a1,sp@-         | save class pointer and new
                movl    d5,sp@-         | index before calling malloc
                movl    #10,sp@-        | push size on stack
                jsr     _malloc
                movl    d0,a0
                addql   #4,sp
                movl    sp@+,d5         | put index back in d5
                movl    sp@+,a1         | put class pointer back in a1
                movw    #0,a0@(0)       | init ref cnt
                movl    a1,a0@(2)       | put class pointer in object
                movl    d5,a0@(6)       | put index into object
                movl    a6@(4),a1       | load return address
                lea     a6@(16),sp      | reset stack ptr, pop AR
                movl    a6@(0),a6       | restore a6
                jmp     a1@(0)          | return to caller
```

Figure 2.1.4: Assembly code for two methods of the class **enum**

Note that **enum** is an abstract type and acts as a template here, i.e. it will be replaced with the actual type name specified by the user. The type **primitive** is not visible to the programmer, it merely represents some internal storage space [Shu91]. Every object of some enumerated type consists of an instance variable *index*, which is an index into the *entryTable* (a table of the string literals as specified in the list of identifiers) and a pointer to the shared table that contains the methods. Integer values starting from zero index the enumerated constants in the list. This allows for meaningful output, should an object of enumerated type receive a message to print itself (method *print*). The index is used to look up the appropriate string literal, which is printed out. It further allows for comparison between enumerated constants since their enumeration implicitly defines an order among them. Along with these logical operations (methods *less*, *lessEqual*, *greater*, *greaterEqual*, *equal*, *notEqual*), the class defines two methods *succ* and *pred*, that return the successor and predecessor, respectively.

```
C12:                                        | new CLASS object, gets
                .long   C12_inst            | assigned a new class number
C12_inst:       .word   1                   | reference count
                .long   C0_shared           | pointer to class CLASS
                .long   0                   | not used
                .long   0                   | not used
                .long   10                  | size of a typical object
                .long   C12_shared          | of this type
C12_numEntries: .long   3                   | three entries
C12_entryTable: .long   C12_helen_literal   | entry table, contains
                .long   C12_leda_literal    | pointers to literals
                .long   C12_zeus_literal
C12_shared:     .long   C12_inst            | pointer to its own class
                .long   C3_inst             | pointer to superclass
                .long   C12_numEntries      | repertoire of methods
                .long   C12_entryTable
                .long   C3_print
                .long   C3_succ
                .long   C3_pred
                .long   C3_less
                .long   C3_lessEqual
                .long   C3_greater
                .long   C3_greaterEqual
                .long   C3_equal
                .long   C3_notEqual
C12_helen_literal:                          | the string literals
                .asciz  "helen"             | pointed to by the
                .align  1                   | entry table
C12_leda_literal:
                .asciz  "leda"
                .align  1
C12_zeus_literal:
                .asciz  "zeus"
                .align  1
```

Figure 2.1.5: Assembly code for type definition of figure 2.1.1

Due to the uniform representation of **Leda** objects in memory, the actual code for the comparison methods of enumerated objects is the same as for objects of type **integer**. An enumerated type defined by the programmer inherits all the methods from the abstract class **enum** and overrides the fields *numEntries* and *entryTable*, since these are specific to each newly defined enumerated type. Figure 2.1.3 shows the predefined assembly code (68000) representing the abstract class **enum**, which could also be viewed as an instance of class CLASS.

Every class is internally identified by a *class number*, in assembly code this number is preceded by the letter 'C'. While C1-C6 are predefined classes, classes higher than C10 are user-defined classes [Shu91]. The fields *numEntries* and *entryTable* have zero entries, since they will be overridden by the actual enumerated type. Note how all comparison operators point to code used by the operators of the predefined class **integer**, which is referred to as C1 (for the representation of class objects in memory see [Shu91]). Figure 2.1.4 shows some more predefined assembly code for the methods **pred** and **succ** of class **enum**. The next figure (2.1.5) shows the assembly code that is created for the type definition in figure 2.1.1. The field *numEntries* now contains the number of enumerated constants and the field *entryTable* has a table of pointers to the literal strings. Figure 2.1.6 illustrates the run-time representation of objects of type **names**.

```
EC_helen:                                   | leda uses pointer semantics
                .long   EC_helen_inst
EC_helen_inst:
                .word   1                   | reference count
                .long   C12_shared          | pointer to its defining class
                .long   0                   | index is 0 for the first object

EC_leda:
                .long   EC_leda_inst
EC_leda_inst:
                .word   1
                .long   C12_shared
                .long   1                   | index is 1 for the second object
```

Figure 2.1.6: Internal representation of the enumerated constants **helen** and **leda**.

2.2 System Defined Enumerated Types

The predefined class **boolean** can be thought of as an enumerated type with the definition in figure 2.2.1. It is the only predefined subclass of the class **enum**, and defines additional methods for the logical connectives *and*, *or* and *not*. The two predefined constants **true** and **false** are shown as **Leda** objects in figure 2.2.2. Figure 2.2.3 shows the memory representation of this class, which is internally named C4. Like every object in the language they consist of a reference count used for garbage collection, a pointer to its shared table and its instance fields (in this case an integer value, that serves as an index into the table of literals) [Shu91].

```
type
  boolean:=(false, true);
```

Figure 2.2.1: An imaginary definition of the predefined type **boolean**

```
EC_false:
                .long   EC_false_inst
EC_false_inst:
                .word   1
                .long   C4_shared           | pointer to its shared table
                .long   0                   | 'false' has index 0
EC_true:
                .long   EC_true_inst
EC_true_inst:
                .word   1
                .long   C4_shared           | pointer to its shared table
                .long   1                   | 'true' has index 1
```

Figure 2.2.2: Predefined enumerated constants **true** and **false** as **Leda** objects

```
C4:
                .long   C4_inst             | pointer to instance table
C4_inst:
                .word   1                   | reference count
                .long   C0_shared           | pointer to metaclass
                .long   0
                .long   0
                .long   10                  | size of a typical object
                .long   C4_shared           | pointer to its shared table
C4_numEntries:
                .long   2                   | two entries
C4_entryTable:
                .long   C4_false_literal
                .long   C4_true_literal
C4_and:
                .long   C4_and_inst         | points to code for the
C4_or:                                      | logical connectives
                .long   C4_or_inst
C4_not:
                .long   C4_not_inst
C4_shared:                                  | shared table of this class
                .long   C4_inst             | pointer to its own class
                .long   C3_inst             | pointer to superclass
                .long   C4_numEntries
                .long   C4_entryTable
                .long   C3_print
                .long   C3_succ
                .long   C3_pred
                .long   C3_less
                .long   C3_lessEqual
                .long   C3_greater
                .long   C3_greaterEqual
                .long   C3_equal
                .long   C3_notEqual
                .long   C4_and              | three new methods for
                .long   C4_or               | the logical connectives,
                .long   C4_not              | that are unique to booleans
```

Figure 2.2.3: Memory representation for the class **boolean**

3 The Logical Paradigm

A basic idea in logic programming is that an algorithm incorporates two parts: the logic and the control. The ideal of logic programming is that the programmer should only have to specify the logic component of an algorithm. The control should be exercised solely by the logic programming system. Unfortunately this ideal has not been achieved with current logic programming systems [Llo84]. This is probably due to the fact, that logic and control are never disjoint components of knowledge. A common-purpose programming language should support both the imperative and logic paradigms in a unified framework [Rad90].

This is certainly also one of the goals of **Leda**. Our belief is, that by providing the imperative (and other) paradigms in addition to the relational one the programmer's task will be significantly enhanced and allow him to more naturally formulate computations than in a single-paradigm language.

3.1 Prolog

Prolog is by far the best known logic programming language and based upon the logic of Horn clauses. The programmer programs by writing assertions, and the Prolog interpreter attempts to validate these assertions, computing values in the process. These assertions can be either facts or rules. Prolog has a built-in facility for deductive retrieval through chronological backtracking and pattern matching via unification. Prolog's declarative style provides a natural way to represent rule-based knowledge. Figure 3.1.1 shows a database defining genealogical information along with an interactive session in Prolog (for a reference of the language see [StS86]).

3.2 Integrating Logical Programming Into Leda

The definition of the logical component in **Leda** is oriented towards the Prolog-style of programming using facts and rules of inference [Bud91a]. While Prolog allows for two undefined variables to be unified (i.e. if either one of the variables is subsequently assigned a value, this will be reflected in both variables), this level of unification is not supported in **Leda** because of the use of call-by-reference parameters, which do not allow for multiple levels of indirections. **Var** parameters are provided in **Leda** next to the usual call-by-value parameters [ShP91].

One of the major objectives in designing the language **Leda** was to keep the language as small as possible. Thus a solution formed out of existing elements of the language would be preferable to the introduction of a new feature [Bud91c]. The original line up of the language included *functions* and *procedures* as the only procedural abstractions, their difference being that a function could also return a value [Bud89c]. During the evolution of the language, the subprogram type *procedure* was abandoned in favor of permitting a *function* to optionally return a value. This decision was made in order to permit for yet another procedural abstraction, the *method*. Unlike a function, a method is only to be used as a class member (instance or shared) in a class definition and its distinctive feature lies in its ability to access the class members [Shu91].

The original idea of integrating logical programming into **Leda** consisted of the introduction of a third procedural abstraction, the *relation*. [Bud91a]. A relation would neither be a function, nor a procedure (or now method), although in imperative code it would act like a procedure and in relational code it could be thought of as a boolean function. Like a function or procedure, a relation would be defined with an argument list and most typically invoked using call-by-reference (or **var**) parameters. Its body would consist of Prolog-style rules either representing facts or rules of inference. A relation is implemented by using *choice points* which will be explained in section 4.

Another idea was the idea of a *generator*, which is an expression that is capable of producing values one at a time on demand. This would allow for backtracking in imperative programming and be accomplished by creating choice points in purely imperative code using the **suspend** statement. Figure 3.2.2 shows an envisioned generator, which generates Fibonacci numbers. It takes one argument that limits the range of the generated Fibonacci numbers. These are passed via the return value of the function, i.e. a **suspend** statement acts like a return statement but in addition places a choice point on the activation record stack, so that the function can be resumed to continue evaluation from where it suspended. The concept of generators was popularized in the language Icon [GrG83].

```
child(helen, leda, zeus).
child(hermione, helen, menelaus).
child(castor, leda, tyndareus).
child(pollux, leda, zeus).
child(aeneas, aphrodite, anchises).
child(telemachus, penelope, odysseus).
child(hercules, alceme, zeus).
mother(Mom, Kid) :- child(Kid, Mom, Dad).

?- mother(Mom, aeneas).
Mom = aphrodite.

Continue (y/n) ? y
NO

?- mother(leda, Kid).
Kid = helen.

Continue (y/n) ? y
Kid = castor.

Continue (y/n) ? y
Kid = pollux.

Continue (y/n) ? y
NO
```

Figure 3.1.1: A genealogical database in Prolog.

While sticking to the implementation proposed for this scheme, we have tried to come up with a solution that does not introduce a new procedural abstraction and yet gives as much flexibility as the original scheme, if not even more. Being inspired by the presence of the **suspend** statement, we have literally merged the above two schemes into one. We decided to use **suspend** statements exclusively for relational programming and allow them inside *functions* and *methods* (the only two subprogram types which are available), giving them meaning in connection with the newly added **rel** parameters.

Generators can generally be expressed as relations, which significantly enhances their usefulness. A generator expressed as a relation can not only return more than one value at a time, but also has the ability to test incoming values for membership. Thus the original semantics of the **suspend** statement as described in [Bud91a] became superfluous, which also adds to the simplicity of the language.

Figure 3.2.3 and 3.2.4 show the new syntax for the programs in figures 3.2.1 and 3.2.3 respectively. Note, that the generator in figure 3.2.4 passes its values now through the formal **rel** parameter b (while still maintaining a pass-by-value parameter n to limit the range of the produced numbers) and not via the return value to the calling subprogram. If the suspend statement is executed, the values of i, j and k are subsequently assigned to the formal **rel** parameter b. Also, the **fail** statement is absent, which will be explained shortly.

```
relation child(var name, mother, father : names);
begin
  child(helen, leda, zeus).
  child(hermione, helen, menelaus).
  child(castor, leda, tyndareus).
  child(pollux, leda, zeus).
  child(aeneas, aphrodite, anchises).
  child(telemachus, penelope, odysseus).
  child(hercules, alceme, zeus).
end;

relation female(var woman : names);
begin
  female(helen).
  female(leda).
  female(hermione).
  female(aphrodite).
  female(penelope).
end;

relation mother(var mom, kid : names);
var
  dad : names;
begin
  mother(mom, kid) :- child(kid, mom, dad).
end;

relation daughter(var lass, parent : names);
begin
  daughter(lass, parent) :- female(lass), mother(parent, lass).
  daughter(lass, parent) :- female(lass), father(parent, lass).
end;
```

Figure 3.2.1: Relations as originally proposed [Bud91a]

```
function fibseq(n : integer)->integer;
var
  count, i, j, k : integer;
begin
  i:=1;
  suspend(i);
  j:=1;
  suspend(j);
  for count:=3 to n do
    begin
      k:=i+j;
      suspend(k);
      i:=j;
      j:=k;
    end;
  fail;
end;
```

Figure 3.2.2: A generator producing the first n Fibonacci numbers [Bud91a]

```
function child(rel name, mother, father : names)->boolean;
begin
  suspend(helen, leda, zeus);
  suspend(hermione, helen, menelaus);
  suspend(castor, leda, tyndareus);
  suspend(pollux, leda, zeus);
  suspend(aeneas, aphrodite, anchises);
  suspend(telemachus, penelope, odysseus);
  suspend(hercules, alceme, zeus);
end;

function female(rel woman : names)->boolean;
begin
  suspend(helen);
  suspend(leda);
  suspend(hermione);
  suspend(aphrodite);
  suspend(penelope);
end;

function mother(rel mom, kid : names)->boolean;
var
  dad : names;
begin
  suspend(mom, kid) :- child(kid, mom, dad);
end;

function daughter(rel lass, parent : names)->boolean;
begin
  suspend(lass, parent) :- female(lass), mother(parent, lass);
  suspend(lass, parent) :- female(lass), father(parent, lass);
end;
```

Figure 3.2.3: Relations as currently implemented

```
function fibseq(n : integer; rel b : integer)->boolean;
var
  count, i, j, k : integer;
begin
  i:=1;
  suspend(i);
  j:=1;
  suspend(j);
  for count:=3 to n
    begin
      k:=i+j;
      suspend(k);
      i:=j;
      j:=k;
    end;
end;
```

Figure 3.2.4: A generator as currently implemented

3.2.1 Rel Parameters

These newly introduced parameters in conjunction with the **suspend** statement are the primary tool of the logical component in **Leda**. These statements allow information to flow out of and into subprograms via unification and backtracking. The information has to be specified in the subprogram definition using the **rel** parameters. These parameters are not specifically defined as input or output parameters and can be used in both manners. They generally act like **var** parameters except that the actual parameter corresponding to a formal **rel** parameter is also allowed to take on an expression [ShP91].

3.2.2 The Suspend Statement

A **suspend** statement can have two forms, which are semantically equivalent to Prolog facts and rules. Both forms take arguments, which have to match the formal parameter list of the **rel** parameters in the subprogram definition. Exhibiting a close resemblance to Prolog, **Leda** subprograms carrying **suspend** statements can be very much thought of as *relations* (with the **suspend** statements themselves being the *clauses*) and are referred to as such below. Unlike Prolog, relations have to return a value of the predefined class **boolean** or some alias. However, arguments to relations can be of any type. Also, **rel** parameters can be mixed with var parameters and value parameters in any desired way. This allows for flexibility in blending the relational paradigm with the other paradigms, as we will shortly see.

A suspend *fact* can be viewed as an assertion about a relation between its arguments. The relation **child** can be viewed as asserting that **helen** is a child of **leda** and **zeus**, all of which are declared to be of the enumerated type **names** (Figure 3.2.2.1). A suspend *rule* describes how new relational information can be derived from existing relations. The *subgoals* on the right-hand side have to evaluate to a value of type **boolean** or some alias, i.e. they can be either other relations or boolean expressions (Figure 3.2.2.2). The order of the subgoals is important and can make a difference, if one is concerned about efficiency. Boolean predicates should generally be used at the end of the list of subgoals, since they are not able to generate new values, but merely can filter out previously generated values (an example will be given in section 5).

From within imperative code a relation is invoked as if it were a boolean function (Figure 3.2.2.3). The returned value may be used or not. Unification causes information to flow in and out across the **rel** parameters. Since the relational paradigm is not manifested by its own procedural abstraction, but rather embedded in functions and methods, arbitrary imperative style code can appear between the clauses. A call on a relation, with some arguments potentially bound to values, is known as a *query*. If a query has multiple solutions one response will be returned.

In order to handle unification, the types of **rel** parameters are required to have a method **equal** defined within their class object. This is already the case for the predefined types **integer** and **real**, as well as for **boolean** and any other enumerated type defined by the programmer. The variables specified as arguments to the **suspend** statements in the examples above don't have to be the formal parameters of the relation. The alternative definition of the relation mother in figure 3.2.2.4 is equivalent to the one in figure 3.2.3.

```
suspend(helen, leda, zeus);
suspend(k);
```

Figure 3.2.2.1: Suspend *facts*

```
suspend(mom, kid) :- child(kid, mom, dad);
suspend(X,Y) :- father(Z,X), father(Z,Y), male(X), X<>Y;
```

Figure 3.2.2.2: Suspend *rules*

```
var
  p1, p2 : names;
  res : boolean;

begin
  p1:=NIL;              // set p1 to 'undefined'
  mother(p1, aeneas); // 'aphrodite' is assigned to p1
  p1.print();           // prints 'aphrodite'

  p2:=NIL;
  mother(aphrodite, p2); // 'aeneas' is assigned to p2
  p2.print();             // prints 'aeneas'

  res:=mother(leda, hercules); // arguments can be expressions
  res.print();                 // prints 'false'
end;
```

Figure 3.2.2.3: Invoking relations from imperative code

```
  relation mother(rel mom, kid : names);
  var
    mommy, baby, dad : names;
  begin
    suspend(mommy, baby) :- child(baby, mommy, dad).
  end;
```

Figure 3.2.2.4: Alternative definition of the relation *mother*

3.2.3 The Fail Statement

To understand failure and backtracking, a discussion of a small amount of the implementation technique is necessary here, although the full technique will be described in section 4.

Whenever the system is faced with the possibility of multiple solutions to a query, a record called *choice point* is pushed on the activation record stack. A choice point has all the information necessary to restore the state of the program to the point where the choice was made. In case of a failure, the most recent choice point is used to reset the system and another alternative is tried. The example in figure 3.2.3.1 demonstrates the use of the **fail** statement in imperative code. It simulates a simple catch/throw control flow. The invocation of the relation *catch* succeeds and unifies the actual parameter with the value **true** and leaves a choice point on the activation record stack. Execution resumes after the query. At some later point in the program (or in some other subprograms) a throw can be made back to the catch routine simply by failing. In imperative code this is accomplished by the **fail** statement. Control is transferred to the last point of choice, which in this case was after the first **suspend** statement in the relation *catch*. The next alternative is tried, which reassigns to the formal parameter *X* the value **false**. Control is then returned once more from the same invocation and a different execution path is followed, since the value of the actual parameter *v* has changed (the program prints out the values 1 and 3).

```
var
  v : boolean;
  i : integer;

  function catch(rel X : boolean)->boolean;
  begin
    suspend(true);
    suspend(false);
  end;

begin
  catch(v);
  if v then
    begin // normal execution
      i:=1; i.print();

      // ... some other code

      fail;
      i:=2; i.print(); // should never get here!
    end
  else
    begin // process caught execution
      i:=3; i.print();
    end;
end;
```

Figure 3.2.3.1: Use of the **fail** statement in imperative code

3.2.4 The Drive Statement

The **drive** statement allows a block of statements to iterate over all solutions of a query. The formal parameter list of the invoked subprogram has to contain at least one **rel** parameter. The relation is invoked with the specified arguments and its most recent choice point (pointing to its next alternative) is saved. The statements specified in the block are executed and after restoring the saved choice point backtracking is invoked by failure. The next alternative will be tried. This scheme will be repeated until the relation finally returns **false** and control is transferred to after the **drive** statement.

Assume the definition of a relation *brother*, that maintains the property that the first argument is a brother of the second argument. The example in figure 3.2.4.1 shows how this relation *brother* might be used with **drive** statements. In order to force the exploration of all pairs that maintain this property the actual parameters of *brother* have to be unbound (i.e. their value is undefined). It is only then that they subsequently get bound to all possible solutions. The second example that uses this relation demonstrates that any expression can be supplied as arguments to relations, thereby reducing the number of solutions.

Nested **drive** statements and the invocation of other relations within the statement block are possible. The only way to break out of a **drive** statement is by returning from within the statement block (assuming that the **drive** statement is used in a subprogram).

By using the **fail** statement control can then be transferred back into the **drive** statement and execution resumes there from the most recent choice point. This is demonstrated in the program in figure 3.2.4.2, which makes use of the generator *fib*, that generates all Fibonacci numbers less than 100. Control jumps back and forth between the main program and the subprogram *generate*. After control is transferred to *generate* for the last time, it will return the value **false** to the main program, thus terminating the **while** loop.

```
drive brother(p1, p2)  // assume that p1 and p2 are undefined
  begin                // prints all pairs of brothers
    p1.print();
    p2.print();
  end;

drive brother(p1, helen) print(p1); // prints all brothers of helen
```

Figure 3.2.4.1 Use of the **drive** statement

```
var
  t : boolean;

  function generate()->boolean;
  var
    k : integer;
  begin
    drive fib(k)
      begin
        k.print();
        if (k=8) | (k=34) then return true;
      end;
    return false;
  end;

begin
  t:=false; t.print();
  while generate()
    begin
      t:=true; t.print();
      fail;
    end;
  t:=false; t.print();
end;

{ output:
false
1
1
2
3
5
8
true
13
21
34
true
55
89
false
}
```

Figure 3.2.4.2: Returning from a **drive** statement

4 Implementing Relations

Leda is implemented as an object-oriented compiler using GNU versions of C++, Lex and Yacc. It produces 68000 assembly language and runs currently on two departmental machines. The code generation part of the **Leda** compiler is described in general in [Shu91]. The implementation of relations is inspired by the David H. D. Warren Abstract Prolog machine (WAM) [MaW88] and explained in [Bud91a]. The following subsections describe the necessary data structures for this implementation technique along with the actual translation of the relational features, that had been introduced in the previous section.

4.1 Choice Points

Because relations are embedded into functions and methods by use of the **suspend** statement, their state is saved by simply leaving the activation record, which holds all the information about the subprogram, on the stack. The code generation for a **suspend** statement then consists of two parts: first a special record is created and pushed on the activation record stack. This record is called a *choice point*. It maintains enough information to restart the system should another alternative be tried. During the second part code is generated that unifies the arguments of the **suspend** statement with the formal **rel** parameters and/or invokes the subgoals on the right hand side (for suspend rules). This code is actual **Leda** code and will be created during parsing.

A choice point consists of 5 pieces of information: the address of the previous choice point (register a3), an instruction address (represented by a label) where execution should resume in the case of backtracking, the current value of the activation record register (register a6), the current value of the environment pointer (register a4) (which is necessary to provide first class functions in **Leda** [Che91]), and register a2 which points to a separate data structure, the *trail stack*.

The trail stack is a stack of addresses, that records the locations of variables that are being assigned during the process of unification. Each choice point records the current location of the trail pointer (register a2). As part of the backtracking process, all values marked above this location are set to the value "undefined". Since **Leda** uses pointer semantics, the 'undefinition' of a variable simply consists of clearing the variable location after decrementing the reference count of the object, that the variable points to, thus facilitating garbage collection.

```
void suspendStatem::makeChoicePt(char *label) {
  //  makeChoicePt -- create a choice point on the stack
  //  label of next address is passed
  comment("make choice point");
  link(reg("a3"), 0, "save old choice pointer, set new one");
  pushea(label, "save next location to branch to on stack");
  push(reg("a6"), "save frame pointer");
  push(reg("a4"), "save environment pointer");
  push(reg("a2"), "save trail pointer");
}
```

Figure 4.1.1: Creating a choice point during code generation

To create a choice point, these pieces of information are pushed on the stack, as illustrated in figure 4.1.1, which shows actual C++ code that is executed during the code generation phase. A pseudo assembly language is used in order to facilitate future portability to different 68000 assembly language dialects.

Because of the use of choice points to save and restore the state of relations, it is important that they are not removed from the activation record stack. This is implemented by a simple change of the subprogram epilogue for every subprogram in **Leda**: The stack pointer (register a6) is never decremented lower than the location pointed to by the current choice pointer (register a3).

Note that this places the responsibility for removing the arguments on the called routine and not---as conventionally done---on the caller. The C++ code for generating a subprogram epilogue is shown in figure 4.1.2. A return statement in the subprogram then simply translates to a jump to this epilogue label. After loading the return address into a register, the current choice pointer is compared to the activation record pointer. If there is a choice point on top of the current activation record, the activation record is not popped off and the jump to the calling routine is issued. In the other case, the frame pointer is restored before the jump is issued. The field *uniqueScopeNum* identifies the scope of a subprogram.

```
void program::genCode()
{
  //  ... other code

  put(str("F", itoa(uniqueScopeNum), "_epi"), "epilogue");
  move(regOff(reg("a6"),4),reg("a1"), "load return address");
  move(reg("a3"), reg("d1"), "move choice point into d1");
  comp(reg("a6"), reg("d1"), "compare frame and choice pointer");
  blt(str("F", itoa(uniqueScopeNum), "_ret"));
  loadea(regOff(reg("a6"), 4*(3 + numParams + !!receiver)), reg("sp"));
  put(str("F", itoa(uniqueScopeNum), "_ret"));
  move(regOff(reg("a6"), 0), reg("a6"), "restore old frame pointer");
  jump(regOff(reg("a1"), 0), "jump back to calling procedure");

  //  ... other code
}
```

Figure 4.1.2: Epilogue of every **Leda** subprogram

4.2 Backtracking And Fail

If backtracking is invoked, several things have to be done: the values of the registers stored in the most recent choice point have to be restored, the effects of any assignments since the previous choice point have to be undone (by setting the variables, whose addresses had been recorded on the trail stack, to the value *undefined*), the stack has to be decremented and a branch to the code location given in the choice point record has to take place. This is all accomplished by the code in figure 4.2.1, which is included in the predefined assembly module. Code generation for backtracking then simply consists of a branch to this code. If no previous choice point is recorded, a run-time message will be issued.

```
_fail:
                movl    a3@(0),d1     | check if choice pointer points
                cmpl    #0,d1         | to a valid choice point
                beq     fail_error
                movl    a3@(-16),d0   | load trail base from choice point
fail_clear:
                cmpl    a2,d0         | compare tp to base
                beq     fail_epi      | branch if reached base
                movl    a2@+, a1      | load address
                cmpl    #0,a1@(0)
                beq     fail_clear    | in case object already undefined
                movl    a1@(0),a0
                subqw   #1,a0@(0)     | decrement refcount
                clrl    a1@(0)        | undefine object
                bra     fail_clear
fail_epi:
                movl    a3@(-4),a1    | get backtrack address
                movl    a3@(-8),a6    | restore frame pointer
                movl    a3@(-12),a4   | restore env pointer
                unlk    a3            | restore choice pointer and sp
                jmp     a1@(0)        | branch to alternate
fail_error:
                pea     fail_error_literal  | error message
                jsr     _strprint
                addql   #4,sp
                movl    #1,sp@-       | return value for program
                jsr     _exit         | program will halt
```

Figure 4.2.1: Code used to restore the system after failure

4.3 Unification

Unification is the basic mechanism, by which **rel** parameters are (repeatedly) assigned values. In general unification consists of matching up two expressions, both of which contain variables. The result is a substitution that, applied to the two expressions, makes them equal.

A unification in **Leda** is slightly different, because a variable can only be unified with an expression. The unification between a variable *var* and an expression *exp* is illustrated by the **Leda** program in figure 4.3.1. This code is actually created in **Leda** and executed during the code generation phase.

If the variable is undefined, the value of the expression is assigned to it. The statement *assign* does the assignment and in addition pushes the address of the variable on the trail stack, enabling a future 'undefinition' of the variable [ShP91]. If the variable is defined, it is compared to the expression. If they are not equal, failure will invoke backtracking. In case they are equal, nothing happens.

Note also that the use of binary operators in a **Leda** program is actually translated into a message, which is sent to the left child of the binary expression [ShP91]. This is why the types of **rel** parameters are required to have a method **equal** defined within their class object. It is used during the unification procedure.

```
function unify(var : varType; exp : expType);
  begin
    if defined(var) then
      if ~(var=exp) then fail // ~ is the not operator
    else
      assign(var, exp);
  end;
```

Figure 4.3.1: A **Leda** program illustrating unification

4.4 Suspend Facts

Suspend statements are implemented through the C++ class *suspendStatem*, whose definition is given in figure 4.4.1. The two constructors in the definition are used for suspend facts and rules respectively. The private field *internalCode* consists of **Leda** code that accomplishes the unification. This code is created during parsing.

Assume the relation in figure 4.4.2, which contains two suspend facts, is parsed. The first constructor of *suspendStatem* will get invoked. Type checking consists of checking the subprogram type (**suspend** statements can only occur within functions or methods), the return type of the subprogram (has to be of **boolean** or some alias) and the correspondence between actual suspend parameters and formal **rel** parameters defined in the subprogram head.

```
class suspendStatem : public Statem { // relational statement
private:
  Statem *internalCode;
public:
  suspendStatem(actualParams*); // for a fact
  suspendStatem(actualParams*, actualParams*); // for a rule
  typeCheckRes checkClauses(actualParams*);
  typeCheckRes checkActualRelParams(typeArgsList*, actualParams*,
    errSuppress=NO_SUPPRESS);
  void makeChoicePt(char*);
  void genCode();
};
```

Figure 4.4.1: The C++ class definition for **suspend** statements

```
  function child(rel name, mother, father : names)->boolean;
  begin
    suspend(helen, leda, zeus);
    suspend(hermione, helen, menelaus);
  end;
```

Figure 4.4.2: The relation *child*

```
        makeChoicePt(R1);
        unify(name, helen);
        unify(mother, leda);
        unify(father, zeus);
        return true;
    R1:
        makeChoicePt(R2);
        unify(name, hermione);
        unify(mother, helen);
        unify(father, menelaus);
        return true;
    R2:
        return false;
```

Figure 4.4.3: Translation of the relation *child*

```
void suspendStatem::genCode()
{
  char s[COMSIZE];
  sprintf(s, "suspendStatem::genCode()");
  comment(s);

  int label;
  makeLabelsR(label,1);
  this->makeChoicePt(str("R", itoa(label)));
  internalCode->genCode(); // generate internal code for relations
  put(str("R", itoa(label)));
}
```

Figure 4.4.4: C++ code for the method *suspendStatem::genCode()*

After type checking is performed, the actual unification code is created and assigned to the field *internalCode*. During the code generation phase the method *suspendStatem::genCode()* is executed (figure 4.4.4.). It first creates a choice point and then generates code for the unification by sending the message *genCode()* to its private field *internalCode*.

A pseudo-code description of this code is shown for two suspend facts in figure 4.4.3, making use of the earlier defined function *unify*. After creating a choice point, the code tests the values of the associated variables, either assigning them or branching to the failure routine. If the unifications succeed the value **true** is returned to the calling program and the activation record is left on the stack, thus enabling future backtracking to this point. If none of the suspend statements of a subprogram can be satisfied, **false** is returned to the calling program after the last statement of the subprogram has been executed. In this case the activation record is popped off.

4.5 Suspend Rules

Suspend rules are a bit more complicated. Their type checking consists of the type checking described above for facts and of an additional check, whether the subgoals evaluate to a boolean expression. After type checking is performed, the arguments of the suspend rule, if they are variables (and not constants), are unified with the incoming actual parameters. This ensures, that information flows to the subgoals on the right hand side.

```
function daughter(rel lass, parent : names)->boolean;
var
  kid, adult : names
begin
  suspend(lass, parent) :- female(lass), mother(parent, lass);
  suspend(kid, adult) :- female(kid), father(adult, kid);
end;
```

Figure 4.5.1: The relation *daughter*

```
    makeChoicePt(R1);
    unify(lass, lass);
    unify(parent, parent);
    if ~female(lass) then fail;
    if ~mother(parent, lass) then fail;
    unify(lass, lass);
    unify(parent, parent);
    return true;
R1:
    makeChoicePt(R2);
    unify(kid, lass);
    unify(adult, parent);
    if ~female(kid) then fail;
    if ~father(adult, kid) then fail;
    unify(lass, kid);
    unify(parent, adult);
    return true;
R2:
    return false;
```

Figure 4.5.2: Translation of the relation *daughter*

In order to demonstrate the code generation for this particular case consider figure 4.5.1, which shows an equivalent definition of the relation *daughter* from figure 3.2.3.

The subgoals are treated like invocations and the backtracking mechanism will take care of the control. The expressions, which must return a value of type **boolean** or some alias, are translated into a conditional statement, which invokes backtracking in case the expression evaluates to **false**. This also explains the fact, that a "return false" statement is implicitly appended to each subprogram, that carries **suspend** statements. Failure from a relation is signalled by returning the value **false**. Figure 4.5.2 shows the pseudo translation for the relation *daughter*.

4.6 Drive

A drive statement generates all possible solutions to a query as the example in figure 4.6.1 shows. Intuitively one might get the same result from within a **Leda** program by executing the code shown in figure 4.6.2 (corresponding to figure 4.6.1). After the first solution has been generated, the **fail** statement transfers control to the most recent choice point, which in this case is the relation *brother* and thus execution resumes from there. A boolean value is again returned from the invocation and tested once more. This loops until all solutions are found and after yielding the last solution the invocation finally returns **false**. Execution then resumes after the **if-then** statement. If there are no solutions to the query, the

statement block of the **then** part will not be executed.

In general however this scheme might not work, since the most recent choice point does not necessarily have to be the one that was created during the invocation of the relation brother, but e.g. a more recent choice point could have been created inside the compound statement by invoking yet another relation. In this case the choice pointer (register a3) might not point to the relation *brother* anymore and thus backtracking will not necessarily immediately return to it.

In order to prevent this from happening, the choice point is saved before executing the compound statement and restored before backtracking is invoked (Figure 4.6.3). Because nested drive statements are allowed, it is not sufficient to store the most recent choice point in a global register. To facilitate proper storage of the current choice pointer, another special stack---next to the trail stack---had to be created, which is called the *expression stack*. Register a5 points to the expression stack; an overview of the register usage in the current **Leda** compiler is given in figure 4.6.6. Figure 4.6.4 shows the C++ class definition of the drive statement and figure 4.6.5 the code for its method *genCode()*.

```
drive brother(p1, p2)
  begin
    print(p1); // prints all pairs of brothers
    print(p2);
  end;
```

Figure 4.6.1: The **drive** statement

```
if brother(p1, p2) then
  begin
    print(p1);
    print(p2);
    fail;
  end;
```

Figure 4.6.2: An intuitive translation of the **drive** statement

```
if brother(p1, p2) then
  begin
    // save current choice point
    print(p1);
    print(p2);
    // restore choice point
    fail;
  end;
```

Figure 4.6.3: The actual translation of the **drive** statement

```
class driveStatem : public Statem {
private:
  invocation *inv;
  Statem     *stats;
public:
  driveStatem(invocation*, Statem*);
  void genCode();
};
```

Figure 4.6.4: C++ class definition of the **drive** statement

```
void driveStatem::genCode()
{
  char s[COMSIZE];
  sprintf(s, "driveStatem::genCode()");
  comment(s);

  int label;

  inv->genCode();
  // code generation of the clause leaves a choice point on the AR stack
  // the choice pointer a3 points to it

  makeLabelsL(label, 1); // get a new label
  compInt(1, regOff(reg("a0"), 6)); // compare result with 'true'
  bne(str("L", itoa(label)));

  pushOnSpecial(reg("a3")); // save choice pointer

  stats->genCode();

  popOfSpecial(reg("a3")); // restore choice pointer
  jump("_fail");

  put(str("L", itoa(label)));
}
```

Figure 4.6.5: The C++ method *driveStatem::genCode()*

```
a0  place for returned value from every code generation routine
a1  auxiliary register
a2  trail pointer
a3  choice pointer
a4  environment pointer
a5  expression stack pointer
a6  frame pointer
a7  stack pointer
```

Figure 4.6.6: Register usage in the implemented compiler

4.7 Further Need For An Expression Stack

Leaving the activation records on the stack along with choice points pointing to them can affect the invocation of subprograms, in the case that relations are invoked during the evaluation of the actual parameters. In conventional compilers, after the value of a parameter has been computed, it is immediately pushed on the activation record stack and the same scheme is applied to the other parameters. This is different in **Leda**. If a relation is invoked during the evaluation of a parameter, its activation record will be left on the stack possibly on top of previously pushed parameters. This can be disastrous, if the parameter values are later retrieved from the invoked subprogram (by using an offset from the static link).

For this reason an intermediate place is needed to temporarily store the parameters. The expression stack, which was made necessary by the implementation of the drive statement can serve this purpose. If a subprogram is invoked, code for the parameters is executed and the computed values are pushed on the expression stack. After the last parameter has been processed, all parameters are copied to the activation record stack, the static link is pushed on the stack and the jump to the subroutine is issued (see the C++ code in figure 4.7.1).

```
void invocation::genCode()
{
  char s[COMSIZE];
  sprintf(s, "invocation::genCode()");
  comment(s);

  if (parameters) {
    parameters->pushOnSpecialStack();
    parameters->copyToRealStack();
  }
  var->genCode(DEREF, INVOKE);
}
```

Figure 4.7.1: Generating code for an invocation

4.8 Rel Parameters

In general, **rel** parameters behave like **var** parameters, i.e. internally their address is passed as actual parameters during an invocation. However, the actual parameters corresponding to **rel** parameters are allowed to take on expressions. The latter is accomplished by creating a dummy pointer to the object, whose address is passed. This prevents changes of the formal parameter from being propagated to the actual parameter, a disastrous feature e.g. if a constant is passed. The code, that pushes the parameters of a subprogram on the expression stack before invoking it, is illustrates in figure 4.8.1.

```
void actualParams::pushOnSpecialStack()
{
  char s[COMSIZE];
  sprintf(s, "actualParams::pushOnSpecialStack()");
  comment(s);

  if (mode == VAR || mode == REL) {

    if (node->isConstant()) { // actually, node is not a variable

      node->genCode(); // generate code for the node
      move(reg("a0"), reg("a1")); // save object in a1
      // create a dummy pointer to this object
      push(reg("a1"));
      pushInt(4); // allocate space for new pointer
      jsr("_malloc");
      addquil(4, reg("sp"));
      pop(reg("a1"));
      move(reg("d0"), reg("a0"));
      move(reg("a1"), regOff(reg("a0"), 0));
      // new dummy pointer is in a1

    } else
      node->genCode(NO_DEREF); // push address if call by reference

  } else { // mode is VAL
    node->genCode();
    incrRefCnt(reg("a0")); // reference count incremented
  }
  pushOnSpecial(reg("a0"));

  if (next) next->pushOnSpecialStack();
  // they are pushed on the special stack in reverse order than they
  // appear later on the AR stack.
}
```

Figure 4.8.1: Generating code for parameters and pushing their values on the stack

5 Multi-Paradigm Programs Using Relations

The following examples illustrate how the relational paradigm and the specific implementation of relations are beneficial to the other paradigms available in **Leda**. Examples are given, that combine the relational with the imperative, functional and object-oriented programming paradigms.

5.1 Blending Relational With Imperative Programming

The program in figure 5.1.1 illustrates the flexibility of choice for formal parameters of a function. The **suspend** statement is only to be used in conjunction with the **rel** parameters. This allows for passing any other information (via either **var** or value parameters) into relations.

```
var
  i : integer;
  k : integer;
  n : real;

  function f(a : real; var b : integer; rel c : integer)->boolean;
  begin
    a:=1.2;
    b:=3;
    suspend(56);
    a:=2.56;
    b:=23;
    suspend(456);
  end;

begin
  // main
  n:=1.5676;
  k:=3;
  i:=NIL;
  drive f(n, k, i)
    begin
      n.print();
      k.print();
      i.print();
    end;
end; // main

{ output:
1.5676
3
56
1.5676
23
456
}
```

Figure 5.1.1: Mixing different types of formal parameters

Any types are allowed for **rel** parameters, provided they carry a method *equal* in their class definition. The program in figure 5.1.2 shows how a genealogical database might be implemented and used in a **Leda** program. Some of the relations appeared in earlier examples. The relation **brother** makes use of the method *notEqual*, which is predefined for enumerated types (note that in **Leda** the operator syntax *X<>Y* is equivalent to *X.notEqual(Y)* or *notEqual(X, Y)*). If the *notEqual* predicate evaluates to the boolean value **false** backtracking will be invoked, forcing the exploration of yet other alternatives. This effectively filters out equal pairs of brothers, that are not desired to be part of the solution. Different examples are given in the main program that show how the relation *brother* might be used in imperative code.

```
type
  names:=(helen, leda, zeus, hermione, menelaus, castor, tyndareus,
          pollux, aeneas, aphrodite, anchises, telemachus, penelope,
          odysseus, hercules, alceme);

var
  p1, p2, p3 : names;
  res : boolean;

  function child(rel name, mother, father : names)->boolean;
  begin
    suspend(helen, leda, zeus);
    suspend(hermione, helen, menelaus);
    suspend(castor, leda, tyndareus);
    suspend(pollux, leda, zeus);
    suspend(aeneas, aphrodite, anchises);
    suspend(telemachus, penelope, odysseus);
    suspend(hercules, alceme, zeus);
  end;

  function male(rel name : names)->boolean;
  begin
    suspend(zeus);
    suspend(menelaus);
    suspend(castor);
    suspend(tyndareus);
    suspend(pollux);
    suspend(aeneas);
    suspend(anchises);
    suspend(telemachus);
    suspend(odysseus);
    suspend(hercules);
  end;

  function mother(rel mom, kid : names)->boolean;
  var
    dad: names;
  begin
    suspend(mom, kid) :- child(kid, mom, dad);
  end;

  function father(rel dad, kid : names)->boolean;
  var
    mom : names;
  begin
    suspend(dad, kid) :- child(kid, mom, dad);
  end;

  function brother(rel X,Y : names)->boolean;
  var
    Z : names;
  begin
    suspend(X,Y) :- father(Z,X), father(Z,Y), male(X), notEqual(X,Y);
    suspend(X,Y) :- mother(Z,X), mother(Z,Y), male(X), notEqual(X,Y);
  end;
```

```
begin // main

  p1:=NIL;
  p2:=NIL;
  drive brother(p1, p2)
    begin                    // prints all pairs of brothers
      p1.print();
      p2.print();
    end;

  p1:=NIL;
  drive brother(p1, helen) p1.print(); // prints all brothers of 'helen'

{ output:
pollux
hercules
castor
pollux
}

  p2:=NIL;
  drive brother(pollux, p2) print(p2); // all siblings of 'pollux'

{ output:
helen
hercules
helen
castor
}
  res:=brother(pollux, hercules)
  print(res);                // prints 'true'

  res:=brother(castor, hercules)
  print(res);                // prints 'false'

end; // main
```

Figure 5.1.2: A genealogical database in **Leda**

5.2 Blending Relational With Functional Programming

The example program in figure 5.2.1 makes use of first class functions and shows how generators interact with each other. The functions *fibseq* and *primeseq* take an integer as an argument and return generators (expressed as relations), that generate all Fibonacci and prime numbers that are less than the specified argument, respectively. These generators can be used either to test numbers for membership, to generate their sequence of numbers or as part of other relations. The relation *fibprime* makes use of the generators *fib* and *prime*, thus constructing a new generator, that produces all numbers less than 250 that are both prime and Fibonacci numbers (these are 2, 3, 5, 13, 89 and 233).

```
type
  generator:=function(rel integer)->boolean;

var
  p : integer;
  fib, prime, fibprime : generator;

  function fibseq(max : integer)->generator;
  begin
    return function(rel n : integer)->boolean;
           var
             i, j, k, count : integer;
           begin
             i:=1; suspend(i);
             j:=1; suspend(j);
             k:=i+j;
             while k<=max
               begin
                 suspend(k);
                 i:=j; j:=k; k:=i+j;
               end;
           end;
  end;

  function primeseq(max : integer)->generator;
  begin
    return function (rel n : integer)->boolean;
           var
             i,k : integer;
             flag : boolean;
           begin
             suspend(2);
             suspend(3);
             flag:=true; k:=5;
             while k<=max
               begin
                 for i:=3 to k/2 if k%i=0 then flag:=false;
                 if flag then suspend(k);
                 flag:=true;
                 k:=k+2;
               end;
           end;
  end;

begin
  fib:=fibseq(250);  // generates all fib numbers less than 250
  prime:=primeseq(250);  // generates all prime numbers less than 250
  fibprime:=function(rel n : integer)->boolean;
            begin
              suspend(n):- fib(n), prime(n);
            end;
  drive fibprime(p) p.print();
end;
```

Figure 5.2.1: The use of generators in **Leda**

5.3 Blending Relational With Object-Oriented Programming

The program in figure 5.3.1 demonstrates how classes may be used in interaction with relations. It models the classical eight queens problem: Eight queens are to be positioned on a chess board in a way that they cannot attack each other.

The class *square* represents the squares, that queens can be positioned on. Next to the *x*- and *y*-coordinates (whose interpretation is obvious), the class has three more methods. The *print* method prints out the coordinates of the receiver queen, the method *equal* checks whether a queen given as argument is equal to the receiver queen, and the method slash returns **false** or **true** depending on whether the receiver queen and the argument queen can attack each other or not, respectively. Note that *slash* and *equal* are actually the names of operators (/ and =, respectively), that are overloaded for class objects here. These operators may be used in infix notation with instances of this class as operands.

This is shown in the relation *eightQueens*. This relation takes eight queens as arguments and verifies, whether they can attack each other or not. It furthermore can generate a solution of the problem, if the queens specified as arguments are undefined (for simplicity the columns of the argument queens p1...p8 are preset, so they can only be moved in their rows by changing the y-coordinate). The relation *valid* checks the validity of its argument queen, i.e. whether it is positioned on the chess board. It will generate a valid square for an argument queen if it has undefined fields. The *slash* predicate will invoke backtracking if any two queens, that are generated so far, can attack each other. Backtracking will then return to the last point of choice, which is in the relation *valid*. Yet another alternative for the recently positioned queen is tried. With the use of the **drive** statement all 92 solutions can be generated.

Note that the method *equal* is required to be defined in the class *square*, since this class is the type of the **rel** parameters of the function *eightQueens*. Omitting to do so will result in a compile-time error.

```
type
  square:=class
    x : integer;
    y : integer;
  shared
    slash : method(square)->boolean;
    equal : method(square)->boolean;
    print : method();
  end;

var
  p1, p2, p3, p4, p5, p6, p7, p8 : square; // represent the eight queens
  res : boolean;

  method square.slash(s : square)->boolean;
  begin
    return (x<>s.x) & (y<>s.y) & (((x-s.x)*(x-s.x))<>((y-s.y)*(y-s.y)));
  end;

  method square.equal(s : square)->boolean;
  begin
    return (x=s.x) & (y=s.y);
  end;

  method square.print();
  begin
    x.print(); y.print();
  end;
```

```
  function valid(rel s : square)->boolean;
  begin
    suspend(s):-col(s.x), row(s.y);
  end;

  function row(rel n : integer)->boolean;
  var
    i : integer;
  begin
    for i:=1 to 8 suspend(i);
  end;

  function col(rel n : integer)->boolean;
  var
    i : integer;
  begin
    for i:=1 to 8 suspend(i);
  end;

  function eightQueens(rel q1, q2, q3, q4, q5, q6, q7, q8 : square)
    ->boolean;
  begin
    suspend(q1, q2, q3, q4, q5, q6, q7, q8):-
      valid(q1),
      valid(q2), q2/q1,
      valid(q3), q3/q2, q3/q1,
      valid(q4), q4/q3, q4/q2, q4/q1,
      valid(q5), q5/q4, q5/q3, q5/q2, q5/q1,
      valid(q6), q6/q5, q6/q4, q6/q3, q6/q2, q6/q1,
      valid(q7), q7/q6, q7/q5, q7/q4, q7/q3, q7/q2, q7/q1,
      valid(q8), q8/q7, q8/q6, q8/q5, q8/q4, q8/q3, q8/q2, q8/q1;
  end;


begin
  // main program, initialize queens, bind them to columns
  p1:=square(1, NIL); p2:=square(2, NIL);
  p3:=square(3, NIL); p4:=square(4, NIL);
  p5:=square(5, NIL); p6:=square(6, NIL);
  p7:=square(7, NIL); p8:=square(8, NIL);

  drive eightQueens(p1, p2, p3, p4, p5, p6, p7, p8)
    begin
      p1.print();
      p2.print();
      p3.print();
      p4.print();
      p5.print();
      p6.print();
      p7.print();
      p8.print();
    end;
  end; // main
```

Figure 5.3.1: The eight queens problem

6 Conclusions

We have succeeded in writing a very usable compiler for the evolving multi-paradigm programming language **Leda**.

Due to the uniform design of objects in the **Leda** compiler, enumerated types are very elegantly treated as a user defined class from inside the compiler. When an enumerated type receives a message to generate code for itself, its class structure is laid out in memory. Type checking of enumerated types is reduced to a match of their respective (unique) class numbers [Che91]. **Leda** objects defined to be of an enumerated type are similar to those of the predefined type **integer**. From an implementational point of view this leads to reuse of predefined methods for the type **integer**. Enumerated constants are gathered during parsing and their code is generated by loading the appropriate label (e.g. "EC_leda") into the address register and dereferencing it if necessary.

The logical programming paradigm, through its declarative style takes on a completely different view of computation. The strength of it lies in its ability to express in a natural way statements that query large bodies of data, while e.g. the imperative paradigm is better suited for complex manipulation of small to moderate amounts of data [Kor86]. We feel that through the redefinition of the semantics of the **suspend** statement and the addition of the **rel** parameters we have come up with a natural blending, where features of both paradigms complement each other in a synergistic way.

One of the deficiencies of Prolog is that the declarative style is clumsy for tasks that are implicitly procedural in nature, such as prompting a user for a series of pieces of information [EvK88]. This is not the case in **Leda** since clauses can be arbitrarily surrounded by imperative code, which is particularly useful for building generators.

In addition to Horn-clauses, Prolog provides several extra logical features which extend its power but also conflict with the classical semantics of logic. These features allow control over the backtracking search strategy and allow clauses to be manipulated, added and deleted from the program during the course of execution. Its most prominent one, the *cut*, is not available in **Leda**, but the manipulation, addition and deletion of clauses can effectively be handled by using parameterized classes [Bud91c], thus once more demonstrating the usefulness of the chosen blend. Negation, a difficult concept in Prolog, can be expressed easily in **Leda**, as the example programs have shown.

The implementation technique (to introduce choice points and leave activation records on the stack) is very efficient and close to many actual Prolog implementations. One drawback is the global nature of choice points. Only the most recent choice point can be reactivated. While this may work well for our chosen example programs, a bigger system might envision the use of local choice points, that are to be defined as **Leda** variables and are manipulated by sending messages to these variables. This would allow for several choice points to coexist.

The **drive** statement explores all alternatives of a query. If generators are involved, it allows to produce all their values by supplying an undefined variable as argument to the invoked relation. There is no elegant method yet to advance a generator, except for the controlled use of the **fail** statement.

Note also that the chosen implementation of relations effectively extends the notion of a generator as proposed in [Bud91a]. A generator in the current implementation (expressed as a relation) can return more than one value at a time, and in addition has the ability to test incoming values for membership.

Another enhancement over the original concept of relations is the fact that arguments to a **suspend** statement within a suspend rule (which are variables) can now be different from the formal **rel** parameters of the relation. This allows for constants to be used in the relation head, a feature that is also available in Prolog. The fact that **var**, **rel** and value parameters can be mixed in any desired way allows for more flexibility in the use of relations.

The **Leda** compiler, as currently implemented, will facilitate further research in the area of multi-paradigm programming languages. The experiences during the implementation of the language **Leda** already led to the refinement of several parts of it. The undertaking of blending different programming paradigms to a linguistic whole in **Leda** is very promising.

References:

[Bud89a] Budd, T.A., "Low Cost First Class Functions", Oregon State University, Technical Report 89-60-12, June 1989. *submitted for publication.*

[Bud89b] Budd, T.A., "Data Structures in **LEDA**", Oregon State University, Technical Report 89-60-17, August 1989.

[Bud89c] Budd, T.A., "The Multi-Paradigm Programming Language **LEDA**", Oregon State University, Working Document, September 1989.

[Bud89d] Budd, T.A., "Functional programming in an Object-Oriented Language", Oregon State University, Technical Report 89-60-16, October 1989.

[Bud91a] Budd, T.A., "**LEDA**: A Blending of Imperative and Relational Programming", *IEEE Software*, January 1991.

[Bud91b] Budd, T.A., "Sharing and First Class Functions in Object-Oriented Languages", Oregon State University, Working Document, March 1991.

[Bud91c] Budd, T.A., "Multi-Paradigm Data Structures in **LEDA**", Oregon State University, Working Document, April 1991.

[Che91] Cherian, V., "Implementation Of First Class Functions And Type Checking For A Multi-Paradigm Language", *Research Paper for M.S. Degree* , Oregon State University, May 1991.

[GrG83] Griswold, R.E., and Griswold, M.T., "The Icon Programming Language", Prentice Hall, Englewood Cliffs, New Jersey, 1983.

[Kor86] Korth, H.F., "Extending The Scope Of Relational Languages", *IEEE Software*, January 1986.

[Llo84] Lloid, J., "Foundations Of Logic Programming", Springer-Verlag, Berlin, 1984.

[MaW88] Maier, D., and Warren, D.S., "Computing With Logic: Logic programming with Prolog", Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1988.

[Rad90] Radensky, A., "Toward Integration Of The Imperative And Logic Programming Paradigms: Horn-Clause Programming In The Pascal Environment", *SIGPLAN Notices* Vol. 25(2):25-34, February 1990.

[ShP91] Shur, J., and Pesch, W., "A **Leda** Language Definition", Oregon State University, Technical Report 91-60-9, September 1991.

[Shu91] Shur, J., "Implementing **Leda**: Objects and Classes", Oregon State University, Technical Report 91-60-11, September 1991.

[StS86] Sterling, L. and Shapiro, E., "The Art Of Prolog", MIT Press, Cambridge, Mass., 1986.