CLEDA – LEDA With Constraint Logic Programming

Masami Takikawa

Department of Computer Science
Oregon State University
Corvallis, Oregon
97331-3202
takikawm@research.cs.orst.edu

Technical Report 93–60–03

March 1993

Abstract

CLEDA is a new programming language descended from the multiparadigm, strongly typed, compiled programming language LEDA. In addition to the four paradigms supported by LEDA, which are imperative, functional, object-oriented, and relational, CLEDA supports the constraint logic programming paradigm. CLEDA is intended to be used to write applications that involve constrained search problems. Constructs provided to support constraint logic programming include:

- Built-in inference engine
  All boolean expressions are "predicates" in the logic programming sense. Built-in operators "`&`" and "`|`" support left-most depth first search and automatic backtracking. Logical expressions can be used in any programming paradigms.
- User definable constraint solver
  Constrained variables of a domain are represented in terms of objects of the corresponding class. Operations and predicates for the domain are written as methods of the class. To restore the necessary information upon backtracking, CLEDA introduces a new built-in operator "`<-`". This operator is similar to the assignment operator "`:=`", but saves the necessary information to be recovered when backtracking occurs.

This paper describes the design and implementation of the language CLEDA.

1 Introduction

CLEDA is a new programming language descended from the multiparadigm, strongly typed, compiled programming language LEDA[ShP91]. In addition to the four paradigms supported by LEDA, which

are imperative, functional, object-oriented, and relational, CLEDA supports the constraint logic programming paradigm.

Placer motivates the idea of multiparadigm languages by saying:

> "A language that supports several paradigms will allow itself to be applied to any given problem domain in a manner most appropriate to that domain." [Pla91]

CLEDA is intended to be used to write applications that involve constrained search problems. For example, compilers involve several constrained search problems, such as the register allocation problem and the label allocation problem. In CLEDA, a compiler writer can use the constraint logic programming paradigm to solve such constrained search problems, use the object-oriented programming paradigm to manipulate a pseudo-code list and symbol tables, and use the imperative programming paradigm to perform file I/O. Each problem can be solved in the most natural way and can easily be combined to build up the whole application.

Constraint logic programming was introduced by Jaffar and Lassez[Jaf87] as a generalization of logic programming. The constraint logic programming (CLP) scheme generalizes the model-theoretic and operational models of logic programming to include constraints over particular problem domains. The scheme explicitly separates inference steps from satisfiability questions.

Most of the existing CLP languages have several constraint solvers built-in. For example, CLP(R)[Jaf87] has built-in constraint solvers in the domains of the reals and finite trees. The major advantage of this approach is that the inference engine can be specialized to deal with domain dependent information. For example, many PROLOG compilers create hash-tables for clause indexing by using information from the domain of finite trees. However, if users want to replace or extend built-in constraint solvers, or add new constraint solvers to the language, there is usually no way to do so.

In contrast, CLEDA does not have any constraint solvers built-in. The system library, users, and/or third parties are to provide constraint solvers. Once a constraint solver for a particular domain is written, it is available to programmers to use on applications that involve problems over that domain. The advantages of this approach are:

- Efficiency
  If the performance of an existing general constraint solver is not satisfactory, the user can replace it with a new specialized constraint solver which is tuned to the type of problems at hand, and hence is faster than the general one.

- Extendability
  If a user wants to solve problems over an unsupported domain, the user can make a constraint solver for that domain and add it to the library.

To achieve this goal, we have divided the CLP part of CLEDA into two portions, the built-in inference engine, which is independent of any particular domains, and the user defined constraint solvers over some specific domains. The built-in inference engine uses a left-most depth first search strategy and utilizes backtracking to retry alternatives. Boolean expressions are regarded as predicates. Conjunctions are represented as binary expressions consisting of two boolean expressions with the built-in operator "`&`" between them. Similarly, disjunctions are binary expressions that use the

built-in operator "|". Boolean expressions, and hence the inference engine, can be used in any of the paradigms.

Constrained variables of a domain are represented in terms of objects of the corresponding class. Operations and predicates for the domain are written as methods of the class. To restore the necessary information upon backtracking, CLEDA introduces a new built-in operator "<-". This operator is similar to the assignment operator ":=", but saves the necessary information to be recovered when backtracking occurs. Consequently, the task of making constraint solvers in CLEDA is actually the task of making classes and methods. In writing the methods, one is free to use any of the programming paradigms to accomplish the task.

The use of the object-oriented paradigm for building constraint solvers makes it easy to use many sorts of domains in one program; operator overloading enables the programmer to use the same operators and predicates over several domains without additional costs to distinguish them at run time.

In March of 1991, we began the project of designing the CLEDA language and implementing a CLEDA compiler. LEDA[Shu91] and CLEDA are independent and have different objectives. Besides supporting constraint logic programming, CLEDA emphasizes portability. The compiler is written in standard C, and produces standard C programs from CLEDA programs. The compiler and generated programs can run on various machines including the IBM-PC, Sequent Balance, HP 9000/433, and Sun4.

This paper describes the design and implementation of CLEDA, concentrating on constraint logic programming. In Section 2, we give a brief review of LEDA and constraint logic programming. We also discuss past and current research related to this work. Section 3 gives a brief introduction to the language with some examples. Section 4 explains implementation of the inference engine and backtracking. Finally, some concluding remarks are given in Section 5.

2 Background

CLEDA is an amalgamation of LEDA and constraint logic programming. Constraint logic programming unifies constraint programming and logic programming. This section discusses those languages to set the scene for this paper.

2.1 Leda

This section provides a brief introduction to programming in LEDA, in an attempt to impart some of the flavor of the language.[1] LEDA is the direct ancestor of CLEDA, so most of CLEDA's syntax and semantics are inherited from LEDA.

LEDA is a strongly typed, compiled programming language. Figure 1 gives a skeleton of a LEDA program. LEDA programs are expressed basically in the PASCAL notation. The choice of PASCAL as a language to be augmented with features supporting multiparadigm programming makes the resulting language, LEDA, simple and accessible to a relatively large audience of programmers.

[1] This introduction describes our implementation of LEDA, which is a revision of the old LEDA described in [ShP91].

```
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 1: Skeleton of a Leda program

Part (a) of Figure 2 shows a simple counting program which makes use of one of the standard control structures provided.[2] `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 (b) of Figure 2 defines a new class `Point` with data variables `x` and `y`. The method `distance` enables objects of the class 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, 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.

All classes have the ability to be invoked by their name as a subprogram, which is the default *constructor* for the class. The constructor may be given an argument for each instance member inherited by or explicitly defined within the class.[3] The order of the parameters must be the same as the order in which the instance members are defined, starting with the inherited members. The constructor dynamically creates a new object, a new instance of that class, and then assigns each parameter to its respective instance variable. The new object is returned. Use of the constructor for class Point can be seen in Figure 2 (b).

LEDA automatically discards objects in those cases where it can be determined that the objects are no longer referenced by any variables, guaranteed to remain thenceforth unused. Variables begin their lives in the formal state of being *undefined*. They can be released from that condition by assigning to them an object other than `NIL`. `NIL` can be used in any expression. It makes the variable or parameter undefined. A special built-in polymorphic predicate `defined(x)` will take any object `x` as an argument and return a boolean indicating its state. If during the course of execution of a LEDA program, an expression attempts to access an instance or shared member via an undefined

[2] These examples are taken from [Shu91].

[3] In the old LEDA, arguments for a constructor are always required, but those arguments are optional in our implementation of LEDA.

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

(b)  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();           // create the Point (0,4)
       p1.x := 0;               // using the constructor without arguments
       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 2: (a) A program that counts to 10; (b) Defining a class, creating new objects, sending a message

```
type
 Link := class:(X)
   datum : X;
   next  : Link:(X);
  shared
   plus  : method(X);
   onEach: method(function(X));
  end;

 List := class:(X)
   data  : Link:(X);
  shared
   init  : method new();
   init2 : method new(X);
   add   : method plus(X);
   append: method plus(List:(X));
   onEach: method(function(X));
   reduce: method:(Z)(Z, function(Z,X)->Z)->Z;
  end;
```

Figure 3: The classes `Link` and `List`

variable, a run-time error occurs.[4]

LEDA supports the basic tools of object-oriented programming—subclassing, inheritance, virtual methods, overriding, and dynamic binding. Not only can methods 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.

Figure 3 shows two examples of *type-parameterized* classes, `List` and `Link`.[5] The class `List` is the visible class accessed by users of the list abstraction. This class merely defines the head of a chain of linked-list values. The class `Link` describes an individual link in this linked-list. Both the classes are type-parameterized, that is, they are qualified by an argument (`X` in both classes) that is unbound at the time the class is defined. To declare an instance of such a class it is necessary for the user to supply a binding for this argument. Examples of such declarations will be provided shortly.

The method `plus` of the class `Link`, which implements adding an element to the end of a list, is shown below:

[4] In the old LEDA, the shared member is taken directly from the statically defined class if the variable is found to be undefined.

[5] This example is adapted from [Bud92].

```
method List.init()
begin
 ;
end;

method List.init2(d:X)
begin
 init();
 self + d;
end;
```

Figure 4: The methods `Init` and `Init2`

```
method Link.plus(d:X)
begin
 if defined(next) then
  next + d
 else
  next := Link:(X)(d);
end;
```

LEDA supports *overloading*; user can redefine the meaning of most of the operators. An expression which uses an operator will be converted into a method call using the method name associated with the operator. For example, the expression `next+d` in the above example will be converted into `next.plus(d)` which calls the method `plus` recursively.

Note that this method (and the other methods described later) can be developed, and code can even be generated for them, without any information regarding the nature of the unbound type `X`.

In the definition of `List` in Figure 3, there are four identifiers after the keyword `method`. Each of these identifiers is called an *alias*. When searching a method, the LEDA compiler first looks at the field names such as `add` and `append` in Figure 3. If the name is not found, the compiler tries to find alias definitions. Aliases are retrieved by those identifiers and formal parameters. For example, consider LEDA compiling the expression `lst+x` where `lst` is a variable of type `List`. The expression is first translated to the method notation `lst.plus(x)`. If the variable `x` is of type `X`, the expression will be the same as `lst.add(x)`. On the other hand, if `x` is a variable of type `List:(X)`, the method `append` will be used instead of `add`.

Users can redefine constructors: if method `new` is found, it will be called after allocating a new object. The method `init` and `init2` in Figure 3 are examples of user defined constructors. The method `init` creates a new empty list when used as a constructor, that is, when the class `List` is called without arguments. Similarly, the method `init2` returns a new list, but it has one argument which is used as an initial member of the list. The definitions of these two methods are shown in Figure 4.

Not only can classes be type-parameterized, but so can functions. An example of type-

```
method List.add(d:X)
begin
 if defined(data) then
  data + d
 else
  data := Link:(X)(d);
end;

method List.append(lst:List:(X))
begin
 lst.onEach(function(d:X) begin self + d; end);
end;
```

Figure 5: The methods `Add` and `Append`

parameterized functions is the method `reduce` in Figure 3. Using `reduce` requires the user to supply a type as a binding for type-parameter `Z`.

LEDA supports *first class functions*, i.e., functions can be used as values and parameters. Both the method `reduce` and `onEach` in Figure 3 receive a function as a parameter. These two methods are defined in Figure 6.

The method `onEach` iterates over the elements of a list. It calls a function parameter, `fn`, for each iteration, passing the element one by one.

The method `reduce` implements a general reduction. The *identity* value, `ident`, is used to "prime the pump" prior to iterating over the elements of the list. This identity may or may not be the same type as the elements in the list. The function parameter, `fn`, takes two values. The left value is of type `Z`, which is the type of identity, and is initially the identity value. The right argument is of type `X`, which is the type of the elements of the list, and ranges over the elements of the list. The function must yield a new value of type `Z`. As each element of the list is examined, this function is invoked, yielding a new value to be used with the next list element. When all list values have been exhausted, the final value of this function is returned.

Figure 7 shows an example use of the `List` class. This program prints the following values:

```
7
24
```

2.2 Constraint Programming

In the imperative programming paradigm, a program consists of a sequence of statements. Computation proceeds by changing environment, namely, the values of variables. Programmers need to specify every dependency of variables. For example, one might write the statement[6]:

[6] This example is taken from [Lel88]

```
method List.reduce:(Z)(ident:Z, fn:function(Z, X)->Z)->Z;
begin
  onEach(function(d:X) begin ident := fn(ident, d); end);
  return ident;
end;

method Link.onEach(fn:function(X));
begin
  fn(datum);
  if defined(next) then
    next.onEach(fn);
end;

method List.onEach(fn:function(X));
begin
  if defined(data) then
    data.onEach(fn);
end;
```

Figure 6: The methods `Reduce` and `OnEach`

```
var aList: List:(integer);
    bList: List:(integer);
    sum : integer;
    prod : real;
begin
  aList := List:(integer)(); aList + 3; aList + 4;
  bList := List:(integer)(2); bList + aList;
  sum := aList.reduce:(integer)(0, integer.plus);
  prod := bList.reduce:(real)(1.0, real.times);
  sum.print();  "\n".print();
  prod.print(); "\n".print();
end;
```

Figure 7: An example use of `List`

```
C := ((F - 32) * 5) / 9;
```

to compute the Celsius(C) equivalent of a Fahrenheit(F) temperature. To convert Celsius temperatures to Fahrenheit, however, a separate statement would have to be included in the program; namely,

```
F := 32 + (9 * C) / 5;
```

along with a conditional statement to choose which statement to execute. To be able to convert temperatures to and from degrees Kelvin, even more statements (with the associated branch points) would have to be added:

```
K := C + 273;
C := K - 273;
K := 290.78 + (5/9)*F;
F := 523.4 + (9/5)*K;
```

As new variables are added to this program, the number of statements grows rapidly.

In contrast, in the constraint programming paradigm, a program consists of a set of relations, called *constraints*, between a set of objects. Programmers need only to write constraints, then a constraint satisfaction system will solve these constraints and print the answer. For example, in the constraint programming paradigm, the statement:

```
C = ((F - 32) * 5) / 9
```

is a program that defines a relationship between degrees Fahrenheit(F) and degrees Celsius(C). Given either F or C, the other can be computed. To add the ability to convert between Kelvin(K) and Celsius(C), only a single additional constraint is required:

```
K = C + 273
```

In addition, a typical constraint satisfaction system can combine these two relationships in order to convert between degrees Kelvin and Fahrenheit without requiring any additional statements.

Large number of problems can be written using constraints. Those include constrained search problems such as the N-Queens problem. Constraint programming makes the programs for solving such problems shorter and easier to write, read, and modify than imperative programming.

Constraint programming languages are usually associated with particular constraint satisfiers. For example, CONSTRAINTS[SuS80] uses local propagation. The associated constraint satisfiers determine the execution speed, defines the computational domains over which programmers can write constraints, and limits what kind of problems can be solved. Local propagation, for instance, cannot solve simultaneous equations.

2.3 Logic Programming

Logic programming is a programming language paradigm in which logical assertions are viewed as programs. There are several logic programming systems in use today, the most popular of which is PROLOG[Bow81]. A PROLOG program is described as a series of logical assertions, each of which is a *Horn clause*. Horn clauses are a subset of the first-order predicate logic, and can be written in the following form:

```
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, alcmene, zeus).
```

Figure 8: A relation consisting only of facts.

```
mother(Mom, Kid) :- child(Kid, Mom, Dad).
father(Dad, Kid) :- child(Kid, Mom, Dad).
grandfather(GF, Kid) :- father(GF, Dad), father(Dad, Kid).
grandfather(GF, Kid) :- father(GF, Mom), mother(Mom, Kid).
```

Figure 9: Relations implementing inference rules.

```
A :- B1, B2, ..., Bn.
```

where `A`, `B1`,..., and `Bn` represent *predicates* (also called *relations*), `n` is greater than or equal to zero (if `n` is zero, the right hand side has no predicates), and the operator `:-` reads "if". In particular, if `n` is zero, the clause is called a *fact* which always holds. Otherwise, the clause is called a *rule* which means that if all of the `B1`,..., and `Bn` are true then `A` is true. The left hand side of a clause is called *head*, and the right hand side is called *body*.

The program is read top to bottom, left to right and search is performed depth-first with backtracking.

For example, one can think of the relation in Figure 8 as a database of genealogical information.[7] The relation `child` consists of a set of facts, and `child(X,Y,Z)` means that `X` is a child of mother `Y` and father `Z`.

Figure 9 shows example rules which use the `child` relation. Identifiers that begin with uppercase letters are variables. Given these definitions, one can ask several questions:

```
?- mother(leda, helen).
yes.
?- mother(X, hermione).
X = helen
?- grandfather(X, Y).
X = hermione
Y = zeus
```

[7] This example is taken from [Bud91a]

where lines that begin with `?-` are user's queries, and the other lines are system's answers.

In order to solve a question, PROLOG looks for a clause whose head matches the question. After matching, the body of the clause is solved instead of the question. In order to find matched clauses, we need a matching procedure that compares two literals and discovers whether there exists a set of substitutions that makes them identical. This procedure is called *unification*, and plays the central role in PROLOG. Thanks to the unification, a relation can be used in several ways. For example, the relation `child` can be used for the verification or search for children or parents as follows:

```
?- child(helen, leda, zeus).  /* verification */
yes.
?- child(Kid, helen, menelaus). /* looking for a child */
Kid = hermione
?- child(castor, Mom, tyndareus). /* seeking for a mother */
Mom = leda
```

2.4 Constraint Logic Programming

In order to solve arithmetic relations in the logic programming paradigm, one needs to use Peano's axioms, i.e., to define predicates using the successor function. The predicate:

```
plus(X, Y, Z)
```

means that `Z` is the sum of `X` and `Y`, and the axioms for addition are:

```
plus(0, Y, Y).
plus(s(X), Y, s(Z)) :- plus(X, Y, Z).
```

Using these axioms, the equation `1+X = 3` is expressed by the query:

```
?- plus(s(0), X, s(s(s(0)))).
```

resulting in the solution `X = s(s(0))`.

The use of the successor function is obviously only feasible for small integers. Programming with Peano's axioms (like coding Turing Machines) is not only anachronistic but incompatible with the very high-level-language characteristics of PROLOG.

In order to avoid the programming difficulty and very slow execution of Peano's axioms, PROLOG has a special built-in arithmetic predicate, `is`. The predicate `is` takes two arguments. The right hand side of the arguments must be an arithmetic expression without unbound variables. The result of the expression will be unified to the left hand side. The above query can be written as:

```
?- X is 3-1.
```

Note that the direct conversion of the above query:

```
?- 1 is X+3.
```

results in an error because the right hand side includes the unbound variable, `X`.

This solution is still unsatisfactory because it breaks the important law in PROLOG that a variable can be used for both the input and output.

Constraint logic programming was introduced by Jaffar and Lassez[Jaf87] as a generalization of logic programming. The constraint logic programming (CLP) scheme generalizes the model-theoretic and operational models of logic programming to include constraints over particular problem domains. In the constraint logic programming paradigm, the body of a clause can include constraints. For example, the query:

```
?- 1 = X+3.
```

is considered as a constraint over the real (or integer) domain. It is the responsibility of the built-in constraint solver to solve this constraint. In this scheme, PROLOG is considered as CLP(finite tree) and unification is considered as a constraint solver for the finite tree domain.

Constraint logic programming languages are very successful. For example, CHIP[Din88] has been used to solve many applications in scheduling, planning, and circuit design.

Most of the existing CLP languages have several constraint solvers built-in. For example, CLP(R)[Jaf87] has built-in constraint solvers in the domains of the reals and finite trees. The major advantage of this approach is that the inference engine can be specialized to deal with domain dependent information. Many PROLOG compilers, for instance, create hash-tables for clause indexing by using information from the domain of finite trees. However, if users want to replace or extend built-in constraint solvers, or add new constraint solvers to the language, there is usually no way to do so.

2.5 Related Work

In this section, we describe some related work. Of course, our language inherits many features from ancestors described in the previous sections. For those languages, consult the above sections as well as the references. In this section, we concentrate on other languages which are not direct ancestors of CLEDA.

2.5.1 The Relational Programming Paradigm in Leda

The original LEDA has its own relational programming paradigm[Pes91] which is totally replaced by the constraint logic programming paradigm in CLEDA.

Figure 10 shows an implementation of the `child` and `mother` relationships in the original LEDA.

The keyword `suspend` is used to define a clause. It acts like the `return` statement, namely, it terminates the enclosed function, but it does not discard the stack frame, which are used later upon backtracking.

The keyword `rel` is used instead of `var`. If the actual parameter for a `rel` formal parameter is a variable, the reference to the variable is passed as the argument. Otherwise, a new variable is created, it gets the value of the actual parameter, and the reference to the new variable is passed as the argument. Thus, constants as well as variables can be passed as reference variables.

The keyword `fail` causes backtracking. The execution returns to the latest choice point, a sequence of the `suspend` statements.

```
type Names := (helen, leda, zeus, hermione, menelaus);

function child(rel Kid, Mom, Dad:Names)
begin
  suspend(helen, leda, zeus).
  suspend(hermione, helen, menelaus).
end;

function mother(rel Mom, Kid:Names)
  var Dad:Names;
begin
  suspend(Mom, Kid) :- child(Kid, Mom, Dad).
end;
```

Figure 10: `Child` and `Mother` relationships in Leda

There are several disadvantages in LEDA's approach.

Firstly, LEDA's built-in unification mechanism is incomplete because the unification depends on the simple assignments to the reference variables. LEDA does not have "logical variables" which are variables that can unify with other variables. Those variables are the essential part of logic languages. The lack of the logical variables is fatal to writing useful logic programs.

Secondly, `suspend` and `fail` approach needs special code to implement them. At the end of every function, a special instruction is used instead of the return instruction. That instruction keeps environment in stack if necessary. Programs written in even the other paradigms need to use this special instruction, hence the execution speed is constrained in every part of the programs. In addition, this special instruction forces the compiler to use assembly language as the object code, in other words, it prevents the compiler from translating LEDA programs into efficient C programs.

Finally, LEDA does not have any way to discard choice points. In PROLOG, cuts(`!`) are used to discard choice points. Cuts are important not only because it constructs "if-then-else" in logic programs (Imagine if LEDA did not have "if-then-else" but only "if-then"), but because it shrinks considerable amount of memory use.

2.5.2 Paslog

PASLOG[Rad90] is a multiparadigm language which integrates an imperative language (PASCAL) and a logic programming language (PROLOG).

Figure 11 shows an implementation of the `child` and `mother` relationships in PASLOG.

The statement `split` creates choice points. The enumerated constant `Any` is used for unbound variables. The function `is` implements unification and is defined in Figure 12.

Like LEDA, PASLOG approach needs special code to keep stack frames at the end of every function. In addition, because assignments are canceled upon backtracking, assignment operators need to save the old content of their left hand side when necessary, and programmers cannot write

```
type Names = (Any, helen, leda, zeus, hermione, menelaus);

function child(var Kid, Mom, Dad:Names):Boolean;
begin
  split
    child := is(Kid, helen) and is(Mom, leda) and is(Dad, zeus);
    child := is(Kid, hermione) and is(Mom, helen) and is(Dad, menelaus);
  end;
end;

function mother(var Mom, Kid:Names):Boolean;
  var Dad:Names;
begin
  Dad := Any;
  mother := child(Kid, Mom, Dad);
end;
```

Figure 11: `Child` and `Mother` relationships in Paslog

```
function is(var x:Names; y:Names):Boolean;
begin
  if x = Any then begin
    is := true;
    x := y;
  end else
    is := (x = y);
end;
```

Figure 12: Unification function `is` in Paslog

programs which use side effects beyond backtracking such as the `setof` predicate.

2.5.3 Prolog in C

J. Boyd and G. Karam[BoK90] have developed a translator of PROLOG into C with the motivation to exploit the complementary nature of procedural and declarative programming; it leads to the objective of a dual PROLOG and C programming environment. The translator accepts PROLOG programs and produces sets of C functions which can be integrated into a C program to form a single, common language target system. Each C function corresponds to a single PROLOG procedure. The emphasis of the translation is placed on producing a consistent and readable format which allows the user to optimize the code by tuning.

Although programmers can write both PROLOG programs which use C functions and C programs which use PROLOG functions, the gap between PROLOG programs and C programs is so big that the programming style needs to be drastically changed.

2.5.4 The Constraint Logic Programming Shell

The Constraint Logic Programming Shell(CLPS)[LiS90] forms the basis for a CLP system that only requires a constraint solver for the desired structure to form a complete CLP implementation. After writing a constraint solver in C for a domain, programmers can write constraint logic programs using that solver.

Although CLPS makes creating CLP systems easier, users cannot introduce additional constraint solvers. Of course, writing constraint solvers and writing CLP programs are completely different tasks which need different skills.

3 The Language CLeda

Our method of integrating LEDA and the constraint logic programming paradigm consists of changing the meaning of binary operators `&` and `|`, and introducing a new operator `<-`. The binary operators `&` and `|` are the basic tool for implementing Horn-clause programs by means of boolean functions. As far as the constraint solver is concerned, it can be implemented via the operator `<-` and the object-oriented feature of LEDA.

3.1 Relational Programming in CLeda

The following simple PROLOG program is a well-known example taken from [Rad90].

```
is_student(john).
likes(mary, pascal).
likes(mary, prolog).
likes(john, prolog).
likes(john, X) :- likes(X, prolog).
may_study(X, Y) :- is_student(X), likes(X, Y).
```

Obviously, the query

```
?- may_study(X, Y).
```

returns the answer:

```
X = john, Y = prolog.
```

We will demonstrate step by step how the above program and the query are implemented in CLEDA.

The following enumerated type declaration is used in the example:

```
type Names := (john, mary, prolog, pascal);
```

The predicates `is_student` and `may_study` are implemented as follows:

```
function is_student(var x:Names)->boolean;
begin
 return eq(x, john);
end;

function may_study(var x,y:Names)->boolean;
begin
 return is_student(x) & likes(x,y);
end;
```

The predicate `eq` implements unification and will be presented later.

Each predicate in PROLOG is implemented as a boolean function. Conjunctions, which appear in the body of clauses, are written as boolean expressions using the operator `&`.

The predicate `likes` is determined by means of four clauses. The operator `|` is used for its implementation:

```
function likes(var x,y:Names)->boolean;
 var z:Names;
begin
 z := prolog;
 return eq(x, mary) & eq(y, pascal) |
        eq(x, mary) & eq(y, prolog) |
        eq(x, john) & eq(y, prolog) |
        eq(x, john) & likes(y, z);
end;
```

Note that the variable `z` is needed because the enumeration constant `prolog` can not be substituted as the formal variable parameter `y` in the call of `likes` above.

Like PROLOG, CLEDA uses a left-most depth-first search strategy, that is, the left hand side of conjunctions and disjunctions will first be examined. Unlike LEDA's `&` and `|`, CLEDA's `&` and `|` supports backtracking.

In general, conversion from Horn-clauses to CLEDA program is a four-step process:

1. Introduce a new boolean function corresponding to the predicate being converted.

2. Make unification explicit.

3. Convert the body of each clause to conjunctive expression using `&`.

4. Combine each conjunctive expression with `|`.

Of course, one can make any combination of `&` and `|` other than a disjunction of conjunctions described above. This flexibility is sometimes useful to write shorter and clearer programs.

The place where such logical expressions can be written is not limited within the `return` statements in boolean functions; logical expressions can be used anywhere boolean expressions are permitted in LEDA. The effect of non-logical control predicate such as `!` in PROLOG is achieved by a set of control statements in LEDA.

The above query can be converted as follows:

```
function main(ac:integer, av:^string)->integer;
 var x,y:Names;
begin
 if may_study(x, y) then begin
   "X = ".print();   x.print();
   ", Y = ".print(); y.print();
   '\n'.print();
 end;
end;
```

If all answers are needed, the `for` statement can be used instead of `if` as follows:

```
function main(ac:integer, av:^string)->integer;
 var x,y:Names;
begin
 for may_study(x, y) do begin
   "X = ".print();   x.print();
   ", Y = ".print(); y.print();
   '\n'.print();
 end;
end;
```

The execution of the program results in the following output:

```
X = john, Y = prolog
X = john, Y = mary
X = john, Y = john
```

The predicate `eq` implements unification in the degree needed in this particular example, namely, it does not support unification of two undefined variables. The predicate `eq` can be written as follows:

```
function eq(var x,y:Names)->boolean;
begin
```

```
  if defined(x) then
    if defined(y) then
      return x = y
    else
      return y <- x
  else
    if defined(y) then
      return x <- y
    else
      return false;
end;
```

The predicate `eq` takes two reference parameters, `x` and `y`, and returns a boolean. If both `x` and `y` hold some values, `eq` compares those values. If one of them is undefined, it gets the value of the other. Otherwise, both parameters are undefined. This case is never used in this example, so `eq` simply returns `false`.

The predicate `eq` acts like two-directional assignment, thus implements unification. For assignment, `eq` uses the operator `<-` instead of usual assignment operator `:=`. Like `:=`, the operator `<-` assigns the value of its right hand side to its left hand side. The difference between those operators is that `<-` saves the old value of its left hand side for backtracking; when backtracking occurs, the assignment is canceled and the old value is restored. The operator `:=` does nothing but assignment, so the new value remains even if backtracking occurs.

3.2 Constraint Logic Programming in CLeda

In this section, we show examples of constraint logic programming in CLEDA. CLEDA does not have any built-in constraint solver; not even as much as the simple unification function `eq` described in the previous section. The system library, users, and/or third parties are to provide constraint solvers.

Suppose we have a simple constraint solver over the integer domain.[8] This constraint solver uses a class, called `Int`, which is used instead of integer. An instance of `Int` can be created by the function `int`, whose prototype is as follows:

```
function int(lower, upper:integer)->Int;
```

The function `int` takes two arguments, `lower` and `upper`. Those arguments define a domain of a constrained variable. For example, in the following program chunk:

```
var i:Int;
begin
  i := int(1, 8);
end;
```

`i` is a constrained variable and ranges over 1 thru 8.

For simplicity, only six methods are available for `Int`, namely, `print`, `select`, `equal`, `notEqual`, `plus`, and `minus`.

[8] The constraint solver described here can be found in the sample files in the Leda distribution.

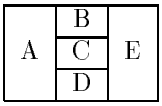

Figure 13: Map coloring problem

The method `print` prints the domain. For example, `i.print()` outputs:

```
1 or 2 or 3 or 4 or 5 or 6 or 7 or 8
```

The method `select` chooses one element from the domain, binds it to the variable, and returns `true`. For instance, `i.select()` returns `true` and as a side-effect `i` becomes `1`. When backtracking occurs, `select` will choose another element. If no more elements are available, `select` returns `false`. The following program prints integer 1 thru 8:

```
var i:Int;
begin
  i := int(1, 8);
  for i.select() do
    i.print();
end;
```

The method `equal` unifies two variables and can be written using the operator `=`. If the variable `j` ranges over 5 thru 15, the unification `i = j` succeeds and both `i` and `j` have a new range 5 thru 8.

The method `notEqual` declares that two variables are different and can be written using the operator `<>`. For example, the expression `i <> j`, where `i=5` and `5<=j<=15`, returns true. As side effects, `j` will now range over `6` thru `15`.

The methods `plus` and `minus` take an integer argument, and add it to (or subtract it from) each element of the domain, and return the new domain. These methods can be written using the operator `+` and `-`. For example, after executing `j := i+5`, where `1<=i<=8`, j will range over `6` thru `13`.

Given the class `Int` and the above methods, let us write some useful programs.

The first program solves the map coloring problem, which is determining the coloring of each area of a map in such a way that any two adjacent areas do not have the same color. In this example, we must paint 5 areas, as shown in Figure 13, using 4 colors.

Figure 14 shows the CLEDA program to solve this problem. (Line numbers are not part of the program. They are for later explanation.)

The program in Figure 14 first defines the domain of all variables (lines 4 and 5). Lines 6 to 9 define all constraints. Then the program assigns actual values for all variables (lines 10 and 11). Finally, if constraints are satisfied, it prints the results (lines 12 to 16).

The selection of values is based on constraints. After the variable `a` is assigned to a value, that value is eliminated from the domains of the variables `b`, `c`, and `d`. In this sense, the program does forward reasoning, and avoids much of the useless backtracking usually found in PROLOG programs.

```
 1  var a, b, c, d, e:Int;
 2  begin
 3    // I have 4 colors.
 4    a := int(1, 4);  b := int(1, 4);  c := int(1, 4);
 5    d := int(1, 4);  e := int(1, 4);
 6    if a <> b  &  a <> c  &  a <> d  &
 7       b <> c  &  b <> e  &
 8       c <> d  &  c <> e  &
 9       d <> e  &
10       a.select()  &  b.select()  &  c.select()  &
11       d.select()  &  e.select()  then begin
12       "a = ".print();  a.print();  "\n".print();
13       "b = ".print();  b.print();  "\n".print();
14       "c = ".print();  c.print();  "\n".print();
15       "d = ".print();  d.print();  "\n".print();
16       "e = ".print();  e.print();  "\n".print();
17    end;
18  end;
```

Figure 14: CLeda program for map coloring

If the keyword `if` is replaced with `for` at line 6, the program will prints all possible answers using backtracking. This backtracking is shallow, that is, one backtracking is enough to get an alternative answer, thus we gain efficiency.

The next example, which uses the same `Int` constraint solver, is the N-Queens problem. The problem is to place N queens on an $N \times N$ chessboard so that no queen can attack another queen. A queen can attack any piece on its diagonals, horizontal row, or vertical column.

The number of solutions of N-Queens as a function of N is given in Table 1.[9] A naive search space, which consists of all possible queen placements, is much larger than the number of solutions. For example, in 9-Queens, the number of possible placements is $9! = 362,880$. The CLEDA program described below uses the constraint technique, which can avoid many placements by exploiting early failure – 9-Queens builds $22,335$ branchpoints which is less than 1% of the naive placements.

First, we show a higher order function which serves as syntactic sugar.

```
function FOR(lower, upper:integer, fn:function(integer)->boolean)->boolean;
begin
  if lower > upper then
    return true;
  return fn(lower) & FOR(lower+1, upper, fn);
end;
```

[9] This table and the number of choice points given below are taken from [Tic91].

N	solutions	N	solutions
4	2	8	92
5	10	9	352
6	4	10	724
7	40	11	2680

Table 1: N-Queens: N vs. number of solutions

The function `FOR` iterates over a range specified by the parameters `lower` and `upper`. It calls a function parameter, `fn`, for each iteration, passing the value of iteration variable. If all calls of that function succeed, `FOR` succeeds. For example,

```
FOR(0, 15, function(i:integer) begin return i >= 0 end);
```

returns `true`, while

```
FOR(-1, 15, function(i:integer) begin return i >= 0 end);
```

returns `false`.

The function `FOR` is an example that shows the power of multiparadigm programming languages. It naturally combines functional programming and constraint programming.

Using `FOR`, we can write the N-Queens program. The main part of the program is shown in Figure 15.

The function `nqueen` takes two parameters: the parameter `n` defines the size of chessboard, and the parameter `printAll` specifies whether or not all answers are necessary. The array `q` holds the location of queens: the index of the array represents the row of a queen, and the value of the array represents the column of a queen.

The function `nqueen` has three sub-programs. The function `queenConstraint` defines constraints of this problem. The function `queenSelect` picks up values which satisfy those constraints. The function `queenPrint` prints an answer.

The function `queenConstraint` is defined as follows:

```
function queenConstraint()->boolean;
begin
  return FOR(1, n,
           function(i:integer)->boolean;
           begin
             return FOR(1, i-1,
                      function(j:integer)->boolean;
                      begin
                        return q[i]<>q[j] & q[i]+i<>q[j]+j & q[i]-i<>q[j]-j;
                      end);
           end);
end;
```

```
function nqueen(n:integer, printAll:boolean);
  var q: array[IntSize] of Int;
      no:integer;
  function queenConstraint()->boolean;
    ...
  function queenSelect()->boolean;
    ...
  function queenPrint();
    ...
  var i:integer;
begin
  if n <= 0 | n > IntSize then begin
    "Parameter is not within 1-".print(); IntSize.print(); "\n".print();
    return;
  end;
  for i := 1 to n do
    q[i] := int(1, n);
  no := 1;
  if printAll then
    for queenConstraint() & queenSelect() do queenPrint()
  else
    if queenConstraint() & queenSelect() then queenPrint();
end;
```

Figure 15: N-Queens program in CLeda

The variable `i` is the row of a queen, and the variable `j` is the row of another queen. The constraints say that the columns of the queens placed at rows `i` and `j` must be different, as must the two diagonals.

The function `queenSelect` is written as follows:

```
function queenSelect()->boolean;
begin
  return FOR(1, n, function(i:integer)->boolean;
                   begin
                     return q[i].select();
                   end);
end;
```

This function simply selects all columns of queens. The selection is based on the constraints described above, thus avoiding all unnecessary choices.

The function `queenPrint` is shown below.

```
  function queenPrint();
    var i:integer;
  begin
    "No. ".print();  no.print();  " : ".print();
    for i := 1 to n do begin
      q[i].print(); ", ".print();
    end;
    "\n".print();
    no := no + 1;
  end;
```

The variable `no` is used for keeping track of the number of answers.

We have shown two examples of constraint-logic programs. Both of them are based on the same simple constraint solver. Once a general constraint solver is written, we can write several constraint-logic programs using it. Constraint-logic programs are easier to write and read, and shorter than programs written in other paradigms for problems in certain domains such as constrained search problems. At the same time, we can gain efficiency due to the elimination of unnecessary choice points.

The map coloring problem is used for register allocation in compilers. The simple constraint solver presented above can be used as a part of compiler which can be developed using multiple programming paradigms supported by CLEDA. We believe large applications such as compilers consist of various problems, and using the right programming paradigm for each problem makes programming much easier.

3.3 The Constraint Solver

In this section, we show how to write a constraint solver using the object-oriented paradigm, employing a subset of the constraint solver described above.

```
type Int := class
    lower, upper : integer;
    bound : Int;
  shared
    new : method(integer, integer);
    equal : method(Int)->boolean;
    print : method();
  end;
```

Figure 16: The definition of the class `Int`

Constraint solvers are written as a CLEDA program which can be included in an application or a library. Programmers can use any programming paradigms supported by CLEDA to write constraint solvers. They do not need to go back and forth between the implementation level and the user level; constraint solvers and CLP programs are written in the same language, CLEDA.

For simplicity, we define the constraint solver for `Int` in such a way that only the methods `equal` and `print` are defined. The definition of class `Int` is shown in Figure 16.

The instance variables `lower` and `upper` specify the range. The instance variable `bound` is used for unification and is described later.

The method `new` is the constructor and is defined as follows:

```
method Int.new(lo, up:integer)
begin
  lower := lo;  upper := up;
end;
```

It simply initializes instance variables according to its parameters. Note that the instance variable `bound` is implicitly initialized to `NIL`, which means undefined.

The method `equal`, which is the heart of this constraint solver, is shown in Figure 17.

If there is no intersection between the constrained variable `self` and `x`, the method `Int.equal` returns `false`. Otherwise, both variables are unified, that is, the two variables become the same, their values being changed to the intersection. The instance variable `bound` is used to combine two constrained variables. If the instance variable `bound` in an instance of `Int` is defined, its value, which is another instance of `Int`, is used instead of that instance. Thus, the actual range of unified variables is stored in one place, the end of `bound` link.

In order to change `lower`, `upper`, and `bound`, the operator `<-` is used instead of normal assignment operator `:=`. As described earlier, the operator `<-` stores the old value, so that the assignment is canceled if backtracking occurs.

The method `print` is shown in Figure 18.

If `bound` is defined, the method `Int.print` prints its value. If the domain consists of only one integer, this integer is printed. Otherwise, two integers with "`..`" between them are printed.

For an example usage of this simple constraint solver, consider the following problem:

```
method Int.equal(x:Int)->boolean;
begin
  if defined(bound) then
    return bound = x;
  if defined(x.bound) then
    return self = x.bound;
  if upper < x.lower | x.upper < lower then
    return false;
  x.bound <- self;
  if lower < x.lower then
    lower <- x.lower;
  if upper > x.upper then
    upper <- x.upper;
  return true;
end;
```

Figure 17: The definition of `Int.equal`

```
method Int.print()
begin
  if defined(bound) then
    bound.print()
  else if lower = upper then
    lower.print()
  else begin
    lower.print(); "..".print(); upper.print();
  end;
end;
```

Figure 18: The definition of `Int.print`

A man `x` is either younger than 25 or older than 50. He can buy beer in Oregon, where people below 21 are not allowed to purchase beer. He has a great-grandchild. He is working for a company, where employees older than 55 must retire. How old is he?

A CLEDA program to solve this problem is shown in Figure 19.

Let us examine the execution step by step. After assignment, the variable `x` gets the range between 0 and 120, which is the possible ages of humans. In the function `young_or_old`, the variable `x` successfully unifies to `young`, and the range of `x` becomes 0 to 24. After calling the function `buy_beer`, the range becomes narrower, between 21 and 24. However, a person of this age cannot have a great-grandchild (here we assume at least 18 of age is necessary to become a parent), so the execution backtracks to the last choice point, in `young_or_old`. In this turn, `x` successfully unifies to `old`, and the range becomes 51 to 120. The function `buy_beer` succeeds without changing the range. After executing `has_greatgrandchild`, the range is narrowed to 54 to 120. Finally, `working` succeeds, making the range between 54 and 55. The program stops after printing the answer, `54..55`.

The above example demonstrated how to make a constraint solver in CLEDA. This constraint solver is exceptionally simple. The constraint solvers used for writing real applications can be much more complicated. However, such effort certainly pays off because we can write many applications using one general constraint solver.

Usually, constraint solvers are built-in. There is no way to change, or add constraint solvers. On the contrary, CLEDA does not have any built-in constraint solvers not even as much as simple unification. We believe the separation of constraint solver from language definition and implementation is very important. This is similar to the separation of I/O library from definition and implementation of C, which allows users to write their own I/O libraries, and makes language definition and implementation simple.

Especially, because constraint solver technology is improving year by year, constraint solvers must be able to be replaced with more efficient ones. In addition, constraint solvers are usually general, so it is important that a user can specialize those for their specific problems in order to gain efficiency.

4 Implementation

Besides supports for constraint logic programming, CLEDA emphasizes portability. The compiler is written in standard C, and produces standard C programs from CLEDA programs. The compiler and generated programs can run on various machines including the IBM-PC, Sequent Balance, HP 9000/433, and Sun4.

The CLEDA compiler converts each function in a CLEDA program into a C function. The arguments are allocated in a heap rather than a stack to be able to implement closures.

This section concentrates on the implementation of the constraint logic programming part of CLEDA, that is, how to implement operators `&`, `|`, and `<-`. Section 4.1 describes the implementation of `&` and `|` which is based on *success continuation passing* technique. The implementation of `<-` is described in Section 4.2.

```
function young_or_old(x:Int)->boolean
var young,old:Int;
begin
  young := Int(0,24);  old := Int(51,120);
  return x = young | x = old;
end;

function buy_beer(x:Int)->boolean;
begin
  return x = Int(21,120);
end;

function has_greatgrandchild(x:Int)->boolean;
begin
  return x = Int(54,120);
end;

function working(x:Int)->boolean;
begin
  return x = Int(16, 55);
end;

var x:Int;
begin
  x := Int(0,120);
  if young_or_old(x) & buy_beer(x) &
     has_greatgrandchild(x) & working(x) then
    x.print()
end;
```

Figure 19: Age guessing program

4.1 Success Continuation Passing

In imperative languages such as C and LEDA, conjunctions can be implemented by the `if` statement. For example, the following function:

```
function foo()->boolean;
begin
  return bar() & baz();
end;
```

can be converted into:

```
function foo()->boolean;
begin
  if bar() then return baz()
           else return false;
end;
```

If the boolean function `bar` succeeds, `foo` returns whatever `baz` returns. It returns `false` otherwise. The execution will never go back into `bar` even if `baz` fails and there are alternatives left in `bar`.

To implement backtracking, we need to make a way to let execution go back to the last choice point. In CLEDA, backtracking is implemented by passing a function closure, which we call *success continuation*, to each boolean function such as `bar`. The success continuation specifies what to do next if the execution of the function succeeds. For instance, the above conjunction will be converted into:

```
function foo(sc:function()->boolean)->boolean;
begin
  return bar(
    function()->boolean
    begin
      return baz(sc);
    end);
end;
```

The parameter `sc` is a success continuation for `foo`. `Foo` will pass a function closure to `bar`. That closure will be executed after `bar` succeeds and will call `baz`. Finally `sc` will be invoked after `baz` succeeds.

After this conversion, we can implement disjunctions. For example, suppose `bar` is defined as follows:

```
function bar()->boolean;
begin
  return xxx() | yyy();
end;
```

It will be converted into:

```
function bar(sc:function()->boolean)->boolean;
begin
  if xxx(sc) then return true
             else return yyy(sc);
end;
```

If both the predicate `xxx` and the success continuation `sc` succeeds, `bar` will succeed. It will retry the alternative, `yyy`, otherwise. As described earlier, the predicate `baz` will be passed in the success continuation for `bar`, so backtracking will occur if `baz` fails.

The `for` statement is also implemented in terms of the success continuation passing. For instance, the following statement:

```
for foo() do "success!\n".print();
```

will be converted into:

```
foo(function()->boolean
    begin
      "success!\n".print();
      return false;
    end);
```

If the predicate `foo` succeeds, the success continuation, which includes the body of the `for` statement, will be invoked. That continuation always returns `false`, so the continuation will repeatedly be called as long as there are successful alternatives.

All the predicates (boolean functions), conjunctions, disjunctions, and `for` statements are converted in the above manner. After this conversion, all uses of `&`, `|`, and `for` will disapear and the resulting program will only use constructs of the original LEDA; no special constructs are necessary to implement backtracking.

4.2 Information Recovering Upon Backtracking

In CLEDA, constraint solvers are implemented in terms of objects and methods as described in Section 3. The state or domain of constrained variables varies as execution proceeds. Those side effects must be canceled when backtracking occurs. To be able to cancel side effects, constraint solvers must use the new assignment operator `<-` instead of the normal assignment operator `:=`. The expression `x<-y` saves the value of `x` and assigns the value of `y` to `x`. If backtracking occurs, the old value will be restored.

The information needed by backtracking is saved in the internal stack called *trail stack*. The pseudo definition of the trail stack is shown in Figure 20.

Each entry of the trail stack consists of the address of a variable and the old value of that variable. The pointer `trailStkPtr` points to the first unused entry and is initialized as follows:

```
trailStkPtr = trailStack;
```

```
typedef struct Trail {
    Address_Of_Object   place;
    Object              oldValue;
} Trail;

Trail trailStack[TrailStkSize];
Trail * trailStkPtr;
```

Figure 20: The pseudo definition of the trail stack written in C

The trail stack grows as the operator `<-` is used and shrinks when backtracking occurs.

The cancellation of assignments is done by the internal function `undo` which is used where there are choice points. The conversion of disjunctions is changed so that `undo` is called. For example, the definition of `bar`:

```
function bar()->boolean;
begin
  return xxx() | yyy();
end;
```

will now be converted into:

```
function bar(sc:function()->boolean)->boolean;
var undoPt : ^Trail;
    success : Boolean;
begin
  undoPt := undoPoint();
  success := xxx(sc);
  undo(success, undoPt);
  if success then return true
             else return yyy(sc);
end;
```

The internal function `undoPoint` returns the current value of `trailStkPtr`. The definition of `undoPoint` is shown in Figure 21. The function `undo` restores the sequence of old values whose range is specified by `trailStkPtr` and `undoPt` if its first parameter is `false`. Therefore, it cancels all assignments made by `<-` during the course of execution of `xxx(sc)` if necessary. The pseudo definition of `undo` is shown in Figure 22.

Because the information stored in the trail stack is unnecessary if there are no choice points, CLEDA does not save the old value in that case. To know whether there is a choice point, CLEDA uses the internal variable called `undoLevel`. It is initialized to zero and incremented by `undoPoint`. Hence there are no choice points if `undoLevel` is zero.

Finally, the operator `<-` is converted to a function call to the internal function `assign` whose definition is shown in Figure 23. If `undoLevel` is not zero, that is, choice points exist, and the old

```
Integer undoLevel;

Trail *undoPoint()
{
    ++undoLevel;
    return trailStkPtr;
}
```

Figure 21: The definition of `undoPoint` written in C

```
void undo(truth, undoPt)
Object truth;
Trail *undoPt;
{
    if (truth is false)
        while (undoPt < trailStkPtr) {
            --trailStkPtr;
            restore trailStkPtr->oldValue to trailStkPtr->place
        }
    --undoLevel;
    if (undoLevel == 0)
        trailStkPtr = trailStack;
}
```

Figure 22: The pseudo definition of `undo` written in C

```
void assign(ptr, newVal)
Address_Of_Object ptr;
Object newVal;
{
    if (undoLevel <= 0)
        assign newVal to the variable pointed to by ptr
    else {
        trailStkPtr->oldValue = the current value of variable
                                pointed to by ptr
        assign newVal to the variable pointed to by ptr
        if (trailStkPtr->oldValue and newVal is not the same object) {
            trailStkPtr->place = ptr;
            trailStkPtr++;
        }
    }
}
```

Figure 23: The pseudo definition of `assign` written in C

value and the new value are not the same object, then `assign` saves the address of the variable and the old value of that variable into the entry pointed to by `trailStkPtr` and increments it.

5 Conclusion

We have designed and implemented a compiler for CLEDA, a direct descendant of multiparadigm, strongly typed, compiled programming language LEDA, augmented with the constraint logic programming paradigm.

CLEDA's constraint logic programming part consists of the built-in inference engine, which is independent of any particular domains, and the user defined constraint solvers over some domains.

The built-in inference engine uses a left-most depth first search strategy and utilizes backtracking to retry alternatives. Boolean expressions are regarded as predicates. Conjunctions and disjunctions are represented as binary expressions with the built-in operator "`&`" and "`|`", respectively. The built-in inference engine is implemented by the success continuation passing technique.

Constrained variables of a domain are represented in terms of objects of the corresponding class. Operations and predicates for the domain are written as methods of the class. The built-in operator "`<-`" is used to restore the necessary information upon backtracking. This operator saves the address of a variable and its old value into the trail stack before writing a new value.

CLEDA can contribute to three different kind of people as follows:

1. application programmers
   In CLEDA, multiple programming paradigms, which are imperative, functional, object-oriented, and constraint logic programming, are available for programmers. They can choose the most

appropriate paradigm to solve a problem, and combine those problems to build the whole application. Especially, CLEDA helps the programmers build applications that involve constrained search problems. Those problems can be easily and efficiently solved in the constraint logic programming paradigm.

2. constraint logic programmers
   Programmers who use only the constraint logic programming paradigm can gain benefit from using CLEDA. In CLEDA, constraint solvers are built upon the object-oriented paradigm. The use of the object-oriented paradigm for building constraint solvers makes it easy to use many sorts of domains in one program; operator overloading enables the programmer to use the same operators and predicates over several domains without additional costs to distinguish them at run time.

3. constraint solver writers
   If a programmer wants to make a constraint logic programming language with his/her own constraint solver, CLEDA can serve as a base language. The programmer needs only to write a constraint solver to make a new constraint logic programming language. Constraint solvers are written as a CLEDA program which can be included in an application or a library. Programmers can use any programming paradigms supported by CLEDA to write constraint solvers. They do not need to go back and forth between the implementation level and the user level; constraint solvers and CLP programs are written in the same language, CLEDA.

In implementing the constraint logic programming part, we attempted to let CLEDA have enough power to be a general purpose programming language. We added new types such as characters and arrays, and developed an I/O library. We also made CLEDA portable among processors; the compiler is written in standard C, and produces standard C programs from CLEDA programs. The compiler and generated programs can run on various machines including the IBM-PC, Sequent Balance, HP 9000/433, and Sun4.

Although the choice of C as the output code made CLEDA portable, the efficiency was sacrificed. CLEDA programs in general run four times slower than the equivalent C programs and ten times slower than the equivalent C++ programs. The most important reason for this is that C is not an appropriate language to implement closures and garbage collection. Because environment for closures cannot be stored in stack, we cannot use the standard C calling sequence; arguments and local variables are stored in a heap rather than a stack. The mark and sweep garbage collection used by CLEDA makes all variables have tags so that garbage collection can know about data types at run time. Research is under way to use a more appropriate intermediate language such as Middleman[Cro92] so that CLEDA can produce better code without sacrificing portability.

6 Acknowledgements

I would like to express my sincere gratitude to my major professor, Dr. Timothy Budd, who was a constant source of ideas and encouragement through the research.

I would also like to thank Dr. Prasad Tadepalli, Dr. Lawrence Crowl, Rajeev K. Pandey and Dr. Hussein Almuallim for their helpful comments and encouragement.

Finally, I would like to thank my wife, Midori Takikawa. Without her patience, understanding, and personal sacrifice, none of this would have been possible.

References

[BoK90] Boyd, J-L., and Karam, G-M., "Prolog in 'C' ", ACM SIGPLAN Notices, Vol.25, No.7, July 1990.

[Bow81] Bowen, D. L., "DECsystem-10 PROLOG USER'S MANUAL", University of Edinburgh, Dept of Artificial Intelligence, Occasional Paper No.27, December 1981.

[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.

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

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

[Bud91c] Budd, T. A., "Avoiding Backtracking by Capturing the Future", Working Document, Oregon State University, March 1991.

[Bud92] Budd, T. A., "Multiparadigm Data Structures in Leda", Proceedings of the 1992 International Conference on Computer Languages, Oakland, CA, April 1992.

[Che91] Cherian, V., "Implementation Of First Class Functions And Type Checking For A Multi-Paradigm Language", *Master's Project*, Oregon State University, May 1991.

[Coh90] Cohen, J., "Constraint Logic Programming Languages", Communications of the ACM, Vol.33, No.7, July 1990.

[Cro92] Crowl, L., "Middleman Language Definition", Working document, Oregon State University, August 1992.

[Din88] Dincbas, P., van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, F., "The Constraint Logic Programming Language CHIP", Proc. FGCS 88, Tokyo, 1988.

[Jaf87] Jaffar, J. and Lassez, J-L., "Constraint Logic Programming", Proc. POPL-87, Munich, 1987.

[Jaf87] Jaffar. J, and Lassez, J-L., "From Unification to Constraints", Proc. Logic Programming 87, Tokyo, June 1987.

[GoR89] Goldberg, A., and Robson, D., *Smalltalk-80: The Language*, Addison-Wesley, Menlo Park, CA, 1989.

[Lel88] Leler, Wm, *Constraint Programming Languages*, Addison-Wesley, Menlo Park, CA, 1988.

[LiS90] Lim, P., and Stuckey, P-J., "A Constraint Logic Programming Shell", Proc. Programming Language Implementation and Logic Programming 90, Linköping, Sweden, August 1990.

[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", ACM SIGPLAN Notices, Vol.26, No.3, March 1991.

[Rad90] Radensky, A., "Toward Integration of The Imperative And Logic Programming Paradigms: Horn-Clause Programming in The Pascal Environment", ACM SIGPLAN Notices, Vol.25, No.2, February 1990.

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

[ShP91] Shur, J., and Pesch, W., "A Leda Language Definition", Oregon State University, Technical Report 91-60-09, September 1991.

[SuS80] Sussman, G., and Steele, G., "CONSTRAINTS – A Language for Expressing Almost-Hierarchical Descriptions", Artificial Intelligence, Vol.14, 1980.

[Tic91] Tick, E., *Parallel Logic Programming*, The MIT Press, Cambridge, MA, 1991.