

Logic Programming in C++

Stephen H. Edwards

Virginia Tech, Dept. of Computer Science

660 McBryde Hall

Blacksburg, VA 24061-0106 USA

+1 540 381 3020

edwards@cs.vt.edu

ABSTRACT

Logic programming is a powerful tool for some problems. To take advantage of this technique when it is appropriate, however, the programming language in use must support it. This paper proposes an approach to implementing logic programming features in C++ through a class library. This approach does not involve integrating C++ with a separate or subordinate Prolog runtime or providing a C++ implementation of a “mini interpreter” for logic rules. Instead, it proposes a natural embedding of logic-style computation in C++ classes. No language modifications or compiler extensions are required to support the library. The implementation strategy is presented, together with examples and a discussion of issues requiring future work.

Categories and Subject Descriptors

D.1.6 [Programming Techniques]: Logic Programming; D.3.2 [Programming Languages]: Language Classifications—*multiparadigm languages*; D.3.3 [Programming Languages]: Language Constructs and Features—*classes and objects*.

General Terms

Multi-paradigm programming.

Keywords

Functional programming, continuation passing, template, programming paradigm, paradigm integration.

1. INTRODUCTION

Logic programming is an approach to software construction that is based on phrasing conditions or constraints in a manner similar to symbolic logic, and then using a logical inferencing process to drive the flow of computation. Logic programming is declarative rather than imperative—as the programmer, you express the form and logical properties of the desired result, rather than provide instructions or an algorithm for how to arrive there. Prolog [6] is perhaps the most well known logic programming language. It has a syntax and semantics radically different from most imperative

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA '02, November 4-8, 2002, Seattle, Washington, USA.
Copyright 2002 ACM 1-58113-000-0/00/0000...\$5.00.

and object-oriented languages in widespread use.

While it is clear that logic programming is not the perfect tool for every software project, there are some application areas where it does excel. Logic programming provides a very powerful paradigm for solving *goal-directed* or *rule-driven* computational tasks. Many planning problems are easy to phrase in terms of goal-directed searching: one lays out a basic framework of elementary moves or steps, then poses a desired goal with the aim of discovering some plan of action (a series of moves) that will achieve the goal. Logic programming is also a natural medium for expressing many symbolic reasoning tasks.

Logic programming brings a different set of benefits (and drawbacks) to a programmer’s toolbox, in comparison to the object-oriented paradigm. Nevertheless, most programmers understandably do not wish to write the majority of their code in a logic programming language. Can logic programming instead be brought into the programmer’s current language and environment, as another tool to use when appropriate?

This paper presents the core of an approach to implementing logic programming features in C++. This approach does not involve simply integrating C++ with a separate or subordinate Prolog runtime or providing a C++ implementation of a “mini interpreter” for logic rules. Instead, it proposes a natural embedding of logic-style computation in a library of classes that can be used for logic programming tasks in C++. No language modifications or compiler extensions are required to support the library. Such a library provides the programmer with the choice of using a logic programming approach when it is the most suitable technique for a given subproblem, while naturally integrating such a solution with other object-oriented code in the context of a larger application.

Section 2 provides some general background on integrating logic programming within other programming paradigms and outlines the solution approach used here. Section 3 presents the central classes that support the approach. Section 4 illustrates how the approach is used with a very simple example, while Section 5 discusses some of the convenience features and add-ons that are possible. Limitations of the current implementation are presented in Section 6. Section 7 briefly outlines related work.

2. TOWARD A SOLUTION

Integrating multiple programming paradigms within one programming language provides several benefits. Budd describes the reasons this way [3]:

Different paradigms arise because problems in disparate areas require diverse approaches to solution, and even in a single area contrasting techniques are often advantageous.

First and foremost, it allows the programmer to choose the “most appropriate tool for the job,” mixing paradigms or solution strategies as he or she sees fits. Second, from an educational point of view, one language offering multiple paradigms may provide a better teaching tool than the traditional approach of switching among several languages to introduce students to new paradigms [4]. Students may be able to focus more of their energy on learning the concepts behind different paradigms rather than being distracted by superficial syntactic differences, execution environments, compilation tools, or error messages.

Because of these benefits, integrating logic programming into C++ has notable potential. The strategy reported here also brings functional programming into the mix, laying the foundation for multiple paradigms within C++ without requiring any changes to the language.

2.1 A Continuation-Passing Approach

Many people who have studied Prolog are familiar with its implementation in terms of Warren’s Abstract Machine (WAM) [5]. Implementing WAM in C++ would lead toward an implementation that had the flavor of a “Prolog interpreter within C++.” Instead, the implementation approach proposed here is based on the functional programming concept of *continuation passing*.

Simply put, a *continuation* is a function that embodies what to do after the current action is completed. A continuation can be thought of as the “future actions” that will take place after the current step in a program. A common functional programming strategy is to use *continuation-passing style*: a function takes a continuation as a parameter that can be explicitly manipulated. This gives the function “control” over how the future actions are invoked, and allows one to build up and manage several possible futures that can be selectively followed. This allows virtually any possible control flow pattern to be established, and is a common approach to providing traditionally imperative concepts (exception handling, short circuit evaluation, abnormal termination, etc.) within a purely functional framework.

Continuations can be used to implement logic programming [5, 8]. While the approach is well proven, it does require first-class support for functional programming in order to work well.

2.2 The FC++ Library

Solid support for functional programming within C++ is a necessary precondition for the continuation-passing approach used here. Fortunately, other researchers have explored ways to support functional programming in C++. One of the most powerful libraries in this arena is the FC++ Library developed by McNamara and Smaragdakis [9, 10]. While the C++ Standard Template Library provides some limited support for functional programming, FC++ provides full support for first class, higher order functions, including parametric polymorphism in the style of Standard ML or Haskell, in a way that is cleanly integrated with the language’s type system. Details on the differences between FC++ and other approaches to functional programming in C++ are discussed elsewhere [9].

In FC++, “functions” are represented by objects that provide an appropriate implementation of the function application operator, `operator()`. Such an object is called a *functoid* in FC++. Any class that provides an appropriate definition for `operator()` together with the necessary type signature information is called a *direct functoid*.

Note that different functoids that provide `operator()` definitions with the same signature need not inherit from the same base class. This introduces a challenge when one wants to create a variable that can range over all functoids with the same signature. FC++ provides a class template for creating *indirect functoids* to solve this problem. An indirect functoid is a clever wrapper similar in implementation to a smart pointer. In effect, it contains a reference to a direct functoid, and can be used to create variables whose values can range over all (monomorphic) functoids with the same signature. The extra level of indirection allows dynamic binding to select the correct implementation of `operator()` based on the actual type of the direct functoid held inside.

3. THE CORE LOGIC IMPLEMENTATION

The central class in the logic implementation is called a `Logic_Relation`. From a functional programming point of view, one can think of a `Logic_Relation` as a boolean predicate that takes one parameter: a continuation representing the remaining logic inference actions to take, presuming the relation will return true.

The use of “relation” in the class name comes from the Prolog view of logic rules as relations. For example, one might define a Prolog-style logic rule called `parent(P, C)`, taking two arguments. Suppose the interpretation given to this rule is that `parent(P, C)` is true if P is the parent of C. Alternatively, we can think of `parent` as a mathematical relation between parents and children. The goal `parent(P, C)` succeeds (that is, returns true) if the pair (P, C) is a member of the relation `parent`, and fails otherwise. Further, logic programming approaches also allow one to systematically iterate over all values for which a relation holds true.

The basic concept behind `Logic_Relation` is simple:

```
class Logic_Relation
{
public:
    bool operator() (
        const Logic_Relation& ) const
    {
        // implementation here
    }
};
```

Here, `operator()` would be used to implement a given logic rule, which would also take as a parameter a continuation representing any future logic rules to pursue. However, because current and future logic rule activations need to be stored and manipulated in building new continuations, the `Logic_Relation` class must be an indirect functoid—basically, a variable that can hold any possible logic rule (or composition of rules).

```

class Logic_Relation : public Fun1< Logic_Relation, bool >
{
public:
    // A copy constructor for initializing from other Logic_Relations
    Logic_Relation( const Logic_Relation& r ) :
        Fun1<Logic_Relation, bool>( (const Fun1<Logic_Relation, bool>&) r ) {}

    // Take a specific direct functoid as an initial value
    template <class DF>
    Logic_Relation( const DF& df ) : Fun1< Logic_Relation, bool >( df ) {}

    // Take a pointer to a direct functoid as an initial value
    Logic_Relation( int x, Impl i ) : Fun1< Logic_Relation, bool >( x, i ) {}
};

// A base class for direct functoids storable in Logic_Relations
class Logic_Relation_Operator : public Fun1Impl< Logic_Relation, bool >
{};

```

Figure 1. Indirect and direct functoids for representing logic relations.

```

class Succeed : public Logic_Relation_Operator
{
public:
    bool operator()( const Logic_Relation& ) const { return true; }
} succeed;

class Fail : public Logic_Relation_Operator
{
public:
    bool operator()( const Logic_Relation& ) const { return false; }
} fail;

```

Figure 2. Functoids for succeed and fail.

```

class Logic_Or : public Logic_Relation_Operator
{
public:
    Logic_Or( const Logic_Relation& left, const Logic_Relation& right ) :
        left_relation (left), right_relation (right) {}

    bool operator()( const Logic_Relation& future ) const
    {
        if ( left_relation( future ) )
            return true;
        else
            return right_relation( future );
    }
private:
    Logic_Relation left_relation;
    Logic_Relation right_relation;
};

inline
Logic_Relation operator || ( const Logic_Relation& left,
                             const Logic_Relation& right )
{
    return Logic_Relation( 0 , new Logic_Or( left, right ) );
}

```

Figure 3. Or'ing together logic relations.

Figure 1 shows the basic definition for `Logic_Relation` as an indirect functoid. The `Fun1` base class is the FC++ template for one-argument indirect functoids. Its parameters specify the type signature of the functoid: in this case, it takes one `Logic_Relation` argument and returns a `bool` result. `Logic_Relation` inherits its `operator()` from this base class. The only definitions added in the `Logic_Relation` class are public constructors used in providing initial values for `Logic_Relation` objects. These constructors simply call their equivalents in the `Fun1` base.

Figure 1 also shows the `Logic_Relation_Operator` class, which is a base class for direct functoids that can be stored in `Logic_Relation` variables. Think of a logic rule—or a `Logic_Relation`—as being a complex logical expression composed of elementary clauses (`Logic_Relation_Operator` instances).

The two simplest logical operators are `succeed` and `fail`. These operators are shown in Figure 2. The `succeed` functoid ignores any continuation passed to it and returns `true`—a “short circuit” success operation. The `fail` functoid provides the dual behavior.

3.1 Supporting Logical Or

Figure 3 shows the code for a “logical or” operator that combines two logic relations. The result will succeed if either of the two component relations succeeds. In Prolog syntax, for example, one might write:

```
parent(P, C) :- father(P, C) ;
               mother(P, C).
```

Here, `P` is the parent of `C` if either `P` is the father of `C` or `P` is the mother of `C`.

The `Logic_Or` class supports this form of combination. It takes two logic relations as constructor parameters and stores them in local data members. Later, when the functoid is evaluated, it will attempt the first relation followed by the given continuation. If this combination fails, it will then attempt the second relation followed by the given continuation. If both alternatives fail, then the combination will fail. For notational convenience, and overloading of `operator||` that takes two logic relations has also been provided.

3.2 Supporting Logical And

Figure 4 shows the code for a “logical and” operator that combines two logic relations. The result will succeed if both of the two component relations succeed. In Prolog syntax, for example, one might write:

```
grandparent(G, C) :- parent(G, P),
                    parent(P, C).
```

Here, `G` is the grandparent of `C` if there is some `P` such that `G` is the parent of `P` and `P` is the parent of `C`.

The `Logic_And` class supports this form of combination. It takes two logic relations as constructor parameters and stores them in local data members. Later, when the functoid is evaluated, it will create a new relation consisting of the first relation followed by the second, then invoke this composition with the given future continuation. For notational convenience, and overloading of `operator&&` that takes two logic relations has also been provided.

3.3 Logic Variables

One key aspect of logic programming is the way logical variables are managed. Logic variables play a role similar to free variables in mathematical formulas. They typically start out being unbound, that is, not associated with any specific value. As application of logic rules proceeds, such a variable may be constrained to a specific value. From this point onward, the variable does have a value. If at this point some logic relation fails, the computation backtracks to an earlier point where an alternative is available (a “choice point” introduced by a logical or combination) and attempts to proceed along an alternate path. Any variable bindings that were made as part of the failed path are reversed or “undone,” causing the corresponding variables to revert back to an unbound state before switching to the alternate path.

The notion of unification and the ability to bind variables and later cancel bindings through backtracking is built into Prolog. In C++, however, logic variables must be implemented from the ground up. Support for logic variables in this approach is provided through a `Logic_Variable` template. This template design is based on a smart pointer design developed by Batov [2].

The `Logic_Variable` template can be thought of as a “smart reference” that can be dynamically bound to a specific value, and that also supports “cancellation” or removal of such a binding at a later time. To support proper unification semantics, logic variables are internally represented as a singly linked chain of references, possibly with a value at the end. Two unbound variables can thus be tied together, so that when one picks up a value later, they both refer to the same object.

Based on Batov’s design, the `Logic_Variable` template contains an elaborate series of template constructors, copy constructors, assignment operators, and cast operations that allow a `Logic_Variable<T>` to contain a reference to an actual value of any subclass of `T` while still behaving properly. Further, a logic variable containing a value of type `T` can be used any place where a `T` object is expected, allowing seamless integration of logic variables into regular C++ code.

Figure 5 outlines the salient client-visible features of the `Logic_Variable` template. Although the full details of the template’s implementation are beyond the scope of this article, complete source code is available from the author.

3.4 Unification

A logic variable that is declared without an initial value is unbound. Attempts to access the value associated with an unbound logic variable raises an exception. Later, an unbound logic variable can be bound (conditionally) to a specific value. A logic variable’s `unify()` operation supports this binding.

In brief, unification in a logic programming language is driven by a logical operator that succeeds if two given items “match.” If the items being matched involve logic variables, these variables are bound to the minimum extent necessary to support a positive match. For unification expressions involving simple logic variables, the process is simple. If both variables have previously been bound to values, the variables match if their values are equal. Alternatively, if one of the variables is unbound, then it can be bound (or tied) to the other by linking their reference chains. If both variables are unbound, they become tied together, even though neither has yet acquired a specific, concrete value.

```

class Logic_And : public Logic_Relation_Operator
{
public:
    Logic_And( const Logic_Relation& left, const Logic_Relation& right ) :
        left_relation( left ), right_relation( right ) {}

    bool operator()( const Logic_Relation& future ) const
    {
        Logic_Relation composite (
            Compose_Continuations( right_relation, future )
        );
    }
private:
    Logic_Relation left_relation;
    Logic_Relation right_relation;

    class Compose_Continuations : public Logic_Relation_Operator
    {
private:
        const Logic_Relation& future1;
        const Logic_Relation& future2;
public:
        Compose_Continuations( const Logic_Relation& f1,
                               const Logic_Relation& f2 ) :
            future1(f1), future2(f2) {}

        bool operator()( const Logic_Relation& ) const
        { return future1( future2 ); }
    };
};

inline
Logic_Relation operator && ( const Logic_Relation& left,
                             const Logic_Relation& right )
{
    return Logic_Relation(0, new Logic_And( left, right ) );
}

```

Figure 4. And'ing together logic relations.

```

template < class Data >
class Logic_Variable
{
public:
    explicit Logic_Variable();
    Logic_Variable( const Data& item );
    Logic_Variable( const Logic_Variable<Data>& ref );
    template < class Other >
    Logic_Variable( const Logic_Variable<Other>& ref );
    // template constructor to initialize Data via a one-arg ctor
    template < class Arg1 >
    Logic_Variable( const Arg1& arg1 );
    // multi-arg and non-const versions are provided as well ...

    operator Data& () const;

    bool is_bound() const;

    class Binding { /* ... */ };

    void Unify( Logic_Variable<Data> rhs, Binding& b, bool& result );
    template < class Other >
    void Unify( Logic_Variable<Other>& rhs, Binding& b, bool& result );
    static void Cancel_Binding( Binding b );

    Logic_Variable< Data >& operator= ( const Logic_Variable<Data>& src );
    template < class Other >
    Logic_Variable<Data>& operator=( const Logic_Variable<Other>& src );

    template < class Other >
    Logic_Variable<Other>& cast();

    // other details elided ...
};

```

Figure 5. The Logic_Variable<T> class template.

```

template <class T>
class Unify : public Logic_Relation_Operator
{
public:
    Unify( const Logic_Variable<T>& left_obj,
          const Logic_Variable<T>& right_obj ) :
        left( left_obj ), right( right_obj ) {}

    bool operator()( const Logic_Relation& future ) const
    {
        typename Logic_Variable<T>::Binding binding;
        bool result;
        Logic_Variable<T> l( left );
        Logic_Variable<T> r( right );

        l.Unify( r, binding, result );
        if ( result && future( succeed ) )
        {
            return true;
        }
        else
        {
            l.Cancel_Binding( binding );
            return false;
        }
    }
private:
    Logic_Variable<T> left;
    Logic_Variable<T> right;
};

template <class T>
Logic_Relation operator |= ( const Logic_Variable<T>& left,
                             const Logic_Variable<T>& right )
{
    return Logic_Relation( 0, new Unify<T>( left, right ) );
}

template <class T1, class T2>
Logic_Relation operator |= ( const Logic_Variable<T1>& left,
                             const T2& right )
{
    return Logic_Relation(
        0,
        new Unify<T1>( left, Logic_Variable<T1>( right ) )
    );
}

```

Figure 6. Unifying logic variables.

```

class Logic_Rule : public Logic_Relation_Operator
{
public:
    virtual Logic_Relation Rule_Definition() = 0;
    bool operator() ( const Logic_Relation& future ) const
    {
        return Rule_Definition()( future );
    }
};

class Parent : public Logic_Rule
{
private:
    // Declare a place to store rule parameters here
    Logic_Variable<string> parent;
    Logic_Variable<string> child;

public:
    // The constructor simply saves away the parameters for later use
    Parent( const Logic_Variable<string>& the_parent,
           const Logic_Variable<string>& the_child ) :
        parent( the_parent ), child ( the_child ) {}

    // This is the real definition of the logic rule
    Logic_Relation Rule_Definition()
    {
        return      Mother( parent, child )
                ||  Father( parent, child );
    }
};

```

Figure 7. User-defined logic rules.

The `unify()` operation supported by `Logic_Variable` returns true for successful unification or matching, and false for a failed match. Successful matches also produce a `Logic_Variable<T>::Binding` object—an encapsulated representation of which internal link, if any, was modified to make the match succeed. This binding can be passed to the logic variable’s `Cancel_Binding()` method later to “break” the tie created by unification.

While `Logic_Variable<T>::unify()` supports the binding of variable values, it does not directly support the inclusion of unification expressions within logical rules since it is not a logic relation. Figure 6 shows a template class for unification expressions. `Unify<T>` takes two logic variables to unify as constructor parameters. When `operator()` is applied, the logic variable’s `unify()` operation is called. On success, the binding is saved while the future continuation is attempted. If the continuation fails, the binding is undone before returning the result to the caller.

For notational convenience in logic expressions, the rarely used `operator|=` has been co-opted to mean unification. Figure 6 shows a template version of this operator that supports concise, natural logic relation expressions.

3.5 User-defined Rules

The preceding subsections lay out the core programming support needed to implement logic programming. In most cases, once a programmer has mastered the library, user code is written in a very different flavor. This section discusses the definition of new

logic rules and Section 3.6 describes how rules are applied or invoked.

Figure 7 presents `Logic_Rule`, which serves as the base class for all user-defined logic rules. This class provides a common definition for `operator()` that provides the correct continuation handling behavior while delegating the real work of the rule to the pure virtual `Rule_Definition()` method. This small addition to the class relieves the end-user from having to continually think about proper management of continuations, allowing one to use logic programming features without requiring a deep understanding of the underlying implementation technique.

Figure 7 also shows a sample logic rule definition for a `Parent` relationship. Each logic rule definition follows the basic pattern of this example, and is divided into three main sections. First, local data members to hold rule parameters are declared. Second, the constructor for the rule simply initializes these data members, “saving away” the logic variables that will be operated on for later access. Third, the rule definition itself appears. The definition for `Parent` shown here presumes that rules for `Mother` and `Father` have already been defined.

3.6 Issuing Goals

Executing, or applying, a logic rule corresponds with issuing a goal in a logic programming language. In C++, this consists of a two-step process. First, one must state the goal, which takes the form of an expression composed from logic relations and logic relation operators. In Figure 7, for example, `Parent::Rule_Definition` contains a goal expression:

```

    Mother( parent, child )
  || Father( parent, child )

```

In the context of the `Parent` rule, this expression represents the “subgoal” to solve in order to determine whether or not the `Parent` rule should succeed. This expression involves two constructor applications, each of which creates a new `Logic_Rule` object. These objects are then combined into a new `Logic_Or` object. This expression does not cause any goal-directed computation to be carried out, however—instead it simply constructs a composite logic relation representing the desired goal.

Second, once such a logic relation object is constructed, it can be evaluated. In this context, evaluation means invoking the relation’s `operator()`, which requires a continuation as an argument. In most cases, to determine if a logic relation is satisfiable, one can simply apply it to the predefined logic relation `succeed`:

```
Parent( P, C )( succeed )
```

This expression will return a Boolean value: true if `P` and `C` are in the relation, and false otherwise. If either `P` or `C` is an unbound logic variable, they will be unified to the extent necessary to satisfy the logic relation. The newly bound value(s) of the variables will be visible after this relation application:

```

if ( Mother( parent, child )( succeed ) )
{
    cout << parent
         << " is the mother of "
         << child << endl;
}

```

4. A SIMPLE EXAMPLE

The user-defined logic rules in Figures 8 and 9 are based on a classic Prolog example from Clocksin and Mellish [6]. They define some basic genealogical rules, supported by some elementary facts about Queen Victoria and a few of her many descendants. First, the `Male` and `Female` relationships define the characters in this example—the universe of discourse for the purposes of the rules shown. Next, a series of rules defining familial relationships appear.

The example rules are self-explanatory and give a flavor of how logic programming in C++ using this framework can be carried out. The `Sibling`, `Aunt`, and `Ancestor` rules show how one can introduce local variables within a rule’s definition to give names to local values. Further, the `Ancestor` rule typifies the way recursive logic inferences are written.

5. ADDITIONAL FEATURES

While Sections 3 and 4 provide the basic foundation for logic programming in C++, there are a number of simple extensions and additions that enhance the usability of the approach. First, it is easy to provide a wider variety of more natural approaches to executing goals. Figure 10 shows a mixin class that can be added as another superclass of both `Logic_Relation` and `Logic_Rule`. This utility class provides an `operator bool()` implementation that forces evaluation of a logic rule. It also provides `for_first()`, `repeat_until()`, and `for_all()`. These template methods take a user-provided functoid as a parameter. In the case of `for_first()` and `for_all()`, this functoid represents some user-defined action to perform on successful completion of the given logic relation. For example, one might pass in a `print_variables` functoid to

cause variable bindings to be printed if the rule concludes successfully. The `for_first()` method finds the first set of logic variable bindings that satisfies the given logic relation. The `for_all()` method systematically enumerates all possible solutions, calling the given functoid once for each distinct solution found.

As indicated by its name, the `repeat_until()` method enumerates solutions to a logic relation until a given condition is achieved. The functoid passed into the method must return a Boolean value. This functoid is invoked after each solution is found, and solutions continue to be enumerated until the functoid returns true or until no more solutions exist. For example:

```

if ( ! Ancestor( A, B ).for_all(
    print_variables( list_with( A, B ) ) )
{
    cout << "no solutions found" << endl;
}

```

It is also possible to add additional relational operators that control the evaluation behavior of logic relation expressions. Figure 11 shows `repeat_until`, which is the equivalent of the `repeat` keyword in Prolog. `repeat_until` takes a logic relation as its argument and produces a new logic relation that forces continual repetition of the incoming logic relation until that relation succeeds. It is only useful when some portion of the logic relation it is given has side effects, and where that relation will eventually change its success conditions through such side effects. This operator is added in two parts. First, the `Logic_Repeat_Until` class defines the type of logic relation produced by a `repeat_until` expression. Next, `Repeat_Until` is a functoid class that takes a logic relation and produces a logic relation is introduced. Finally, `repeat_until` is defined as an instance of the `Repeat_Until` class, serving as a higher-order logic functoid that can be directly applied in expressions.

6. ISSUES FOR FUTURE EXPLORATION

The logic programming implementation strategy presented here provides a simple but capable logic programming framework in C++. Further, building logic programming on top of the FC++ library provides a true multi-paradigm programming platform that supports imperative, object-oriented, functional, and logic programming styles. At the same time, however, there are several aspects of the approach that deserve further attention.

First, the discussion of unification presented in this paper only addresses logic variables. FC++ includes a powerful, optimized list abstraction that supports lazy lists. This FC++ facility is modeled on Haskell’s lazy lists. To support logic programming more completely, unification needs to be extended to other structures, including lists. Further, a practical strategy for extending unification support to user-defined objects is also essential.

```

class Male : public Logic_Rule
{
private:
    Logic_Variable<string> person;
public:
    Male( const Logic_Variable<string>& the_person ) :
        person( the_person ) {}

    Logic_Relation Rule_Definition()
    {
        return ( person |= "Albert" ) || ( person |= "Edward" ) || ( person |= "George" );
    }
};

class Female : public Logic_Rule
{
private:
    Logic_Variable<string> person;
public:
    Female( const Logic_Variable<string>& the_person ) :
        person( the_person ) {}

    Logic_Relation Rule_Definition()
    {
        return ( person |= "Alice" ) || ( person |= "Mary" ) || ( person |= "Victoria" );
    }
};

class Father : public Logic_Rule
{
private:
    Logic_Variable<string> father;
    Logic_Variable<string> child;
public:
    Father( const Logic_Variable<string>& the_father,
            const Logic_Variable<string>& the_child ) :
        father( the_father ), child( the_child ) {}

    Logic_Relation Rule_Definition()
    {
        return ( ( father |= "Albert" ) && ( child |= "Edward" ) )
            || ( ( father |= "Albert" ) && ( child |= "Alice" ) )
            || ( ( father |= "Edward" ) && ( child |= "George" ) );
    }
};

class Mother : public Logic_Rule
{
private:
    Logic_Variable<string> mother;
    Logic_Variable<string> child;
public:
    Mother( const Logic_Variable<string>& the_mother,
            const Logic_Variable<string>& the_child ) :
        mother( the_mother ), child( the_child ) {}

    Logic_Relation Rule_Definition()
    {
        return ( ( mother |= "Victoria" ) && ( child |= "Edward" ) )
            || ( ( mother |= "Victoria" ) && ( child |= "Alice" ) )
            || ( ( mother |= "Mary" ) && ( child |= "George" ) );
    }
};

```

Figure 8. Geneology rules and Queen Victoria.

```

class Sibling : public Logic_Rule
{
private:
    Logic_Variable<string> sib1, sib2;
public:
    Sibling( const Logic_Variable<string>& the_sib1,
            const Logic_Variable<string>& the_sib2 ) :
        sib1( the_sib1 ), sib2( the_sib2 ) {}

    Logic_Relation Rule_Definition()
    {
        Logic_Variable<string> mother, father;
        return Father( father, sib1 ) && Father( father, sib2 )
            && different( sib1, sib2 )
            && Mother( mother, sib1 ) && Mother( mother, sib2 );
    }
};

class Aunt : public Logic_Rule
{
private:
    Logic_Variable<string> aunt, nephew;
public:
    Aunt( const Logic_Variable<string>& the_aunt,
          const Logic_Variable<string>& the_nephew ) :
        aunt( the_aunt ), nephew( the_nephew ) {}

    Logic_Relation Rule_Definition()
    {
        Logic_Variable<string> parent;
        return Female( aunt ) && Sibling( aunt, parent ) && Parent( parent, nephew );
    }
};

class Grandparent : public Logic_Rule
{
private:
    Logic_Variable<string> grandparent, child;
public:
    Grandparent( const Logic_Variable<string>& the_grandparent,
                const Logic_Variable<string>& the_child ) :
        grandparent( the_grandparent ), child ( the_child ) {}

    Logic_Relation Rule_Definition()
    {
        Logic_Variable<string> parent;
        return Parent( grandparent, parent ) && Parent( parent, child );
    }
};

class Ancestor : public Logic_Rule
{
private:
    Logic_Variable<string> ancestor, descendant;
public:
    Ancestor( const Logic_Variable<string>& the_ancestor,
              const Logic_Variable<string>& the_descendant ) :
        ancestor( the_ancestor ), descendant( the_descendant ) {}

    Logic_Relation Rule_Definition()
    {
        Logic_Variable<string> child; // Ancestor's child
        return Parent( ancestor, descendant )
            || ( Parent( ancestor, child ) && Ancestor( child, descendant ) );
    }
};

```

Figure 9. Additional genealogy rules.

```

template < class LR_Base >
class Utilities_Mixin_For
{
public:
    virtual bool operator() ( const LR_Base& ) const = 0;
    operator bool() const { return operator()( succeed ); }

    template < class DF >
    bool for_first( const DF& df ) { return operator()( For_First_DF< DF >( df ) ); }

    template < class DF >
    bool repeat_until( const DF& df ) { return operator()( Until_DF<DF>( df ) ); }

    template < class DF >
    bool for_all( const DF& df )
    {
        For_All_DF<DF>* df_wrapped = new For_All_DF<DF>( df );
        Logic_Relation df_wrapper( 0, df_wrapped );
        operator()( df_wrapper );
        return df_wrapped->triggered; // df_wrapped is cleaned up when df_wrapper destructs
    }

private:
    template < class DF >
    class For_First_DF : public Logic_Relation_Operator
    {
        const DF& df;
    public:
        For_First_DF( const DF& the_df ) : df( the_df ) {}
        bool operator()( const Logic_Relation& ) const { df(); return true; }
    };

    template < class DF >
    class Until_DF : public Logic_Relation_Operator
    {
        const DF& df;
    public:
        Until_DF( const DF& the_df ) : df( the_df ) {}
        bool operator()( const Logic_Relation& ) const { return df(); }
    };

    template < class DF >
    class For_All_DF : public Logic_Relation_Operator
    {
        const DF& df;
    public:
        bool triggered;
        For_All_DF( const DF& the_df ) : df( the_df ), triggered( false ) {}
        bool operator()( const Logic_Relation& ) const
        {
            // record that at least one case worked
            bool& trig = const_cast< bool& >( triggered );
            trig = true;
            df();
            return false; // To force further iterations
        }
    };
};

class Logic_Relation : public Fun1< Logic_Relation, bool >,
                    public Utilities_Mixin_For< Logic_Relation >
{
    // ...
};

```

Figure 10. A mixin class adding features to logic relations.

```

class Logic_Repeat_Until : public Logic_Relation_Operator
{
public:
    Logic_Repeat_Until( const Logic_Relation& r ) : relation( r ) {}

    bool operator() ( const Logic_Relation& future ) const
    {
        while ( ! relation( succeed ) )
        {
            // loop until it succeeds, then proceed
        }
        return future( succeed );
    }

private:
    Logic_Relation relation;
};

class Repeat_Until : public CFunType< Logic_Relation, Logic_Relation >
{
public:
    Logic_Relation operator () ( const Logic_Relation& relation ) const
    {
        return Logic_Relation( 0, new Logic_Repeat_Until( relation ) );
    }
} repeat_until;

```

Figure 11. Adding a `repeat_until` operator.

Second, this approach to implementing logic programming relies heavily on delayed evaluation. Expressions that form logic relations build an object structure that represents a query. Query execution happens later, however, when the corresponding `operator()` is invoked. Because continuations are evaluated selectively, only some of the subexpressions in a goal might actually be executed in any given query application. This behavior is familiar to those with logic programming experience, but directly conflicts with the strict evaluation policy of C++. C++, like most imperative and object-oriented languages, evaluates all arguments before calling an operation. This “eager evaluation” strategy makes it difficult to simply insert plain C++ code fragments directly inside logic relation expressions. C++ code fragments inserted this way are evaluated as the logic relation is being built, rather than being delayed and only evaluated conditionally as the relation is being invoked later. An approach to providing cleanly integrated *promises* (that is, delayed computations) would make it easier to embed C++ code actions within logic relation expressions.

Third, the functoid continuations used in this C++ implementation are limited in scope to one (arbitrarily large) logic relation invocation. As a result, there are implicit boundaries to how far backtracking can retreat in searching for a logic solution. Each logic expression has an implicit backtracking boundary, and execution within the context of a logic expression proceeds somewhat differently than normal, straight-line C++ code. This results in a small impedance mismatch between non-logic-based code sequences that contain isolated logic queries. In a language like Scheme that supports `call-with-current-continuation`, it is possible to achieve a total embedding of the logic execution features with the language’s native model of

computation. The C++ solution here is reminiscent of ICON in this respect—semicolons (and other locations) represent checkpoints across which backtracking may not cross.

Finally, the work reported here was developed using g++ version 3.x, and has also been checked on v.2.95.2. Because of the requirements of FC++ and the heavy use of template methods, including template methods nested within template classes, the current code implementation requires a compiler that supports these features.

7. RELATED WORK

The approach described here relies heavily on FC++ and a well-known technique for implementing logical inference. The contribution lies instead in demonstrating the feasibility and practicality of packaging this approach in a generally reusable class library. A comparison of the functional programming approach used in FC++ with other approaches to supporting first-class functions through objects is available elsewhere [9].

A number of other researchers are actively investigating multi-paradigm languages. Budd developed Leda specifically as a multi-paradigm language for teaching [4]. The Oz language supports functional and logic programming in a concurrent, object-oriented framework [11, 12]. LIFE (Logic, Inheritance, Functions, and Equations) aims to provide clean integration of constraint logic programming and functional programming with solid object-oriented foundations [1]. Hanus provides a detailed discussion of the state of the practice in integrating functional programming with logic programming [8].

In contrast to this prior work, instead of developing a new, multi-paradigm language, the focus here is on adding new paradigms to an existing language without requiring any language modifica-

tions or compiler extensions. While it is possible to provide cleaner, more syntactically pleasing results by starting from scratch, it is unlikely that such efforts toward new language development will see the same scope of installed user base as C++ has at any point in the near future. To maximize the potential for benefiting practicing programmers (and programmers in training), it is helpful to focus on languages currently in use.

8. CONCLUSIONS

There are positive benefits to be gained by supporting multiple programming paradigms in one language. It broadens the programmer's selection of available solution techniques. It may also simplify the process of learning alternative programming paradigms. This paper presents the core of an approach to implementing logic programming features in C++ through a library of classes and templates based on the FC++ library. No language modifications or compiler extensions are required to support the approach. Packaging logic programming services in this way provides the programmer with the choice of using a logic programming approach when it is the most suitable technique for a given subproblem. It also provides a natural path for integrating such a solution with other object-oriented code in the context of a larger application. While many programmers may have considered logic programming to be antithetical to a language like C++, it is clearly feasible to provide native support for this paradigm.

9. ACKNOWLEDGMENTS

I gratefully acknowledge the contributions to this work provided by David McPherson, who worked on the initial logic programming implementation, and Mike O'Laughlen, who has helped refine some of the implementation ideas and is currently working on experimental evaluation. Complete source code for this article, including examples, is available by sending e-mail to the author.

10. REFERENCES

[1] Ait-Kaci, H. and Podelski, A. Towards a meaning of LIFE. Technical Report PRL-RR-11, School of Computing Science, Simon Fraser Univ., 1993, available on-line: < <http://www.isg.sfu.ca/life/>>

- [2] Batov, V. Safe and economical reference counting in C++. *C/C++ Users Journal*, June 2000.
- [3] Budd, T.A. Blending imperative and relational programming. *IEEE Software*, Jan. 1991, 8(1): 58-65.
- [4] Budd, T.A. *Multiparadigm Programming in Leda*. Addison Wesley, Reading, Massachusetts, 1995.
- [5] Campbell, J.A., ed. *Implementations of Prolog*. Ellis Horwood, 1984.
- [6] Clocksin, W.F., and Mellish, C.S. *Programming in Prolog*. Springer-Verlag, Berlin, 1981.
- [7] Griswold, R.E., and Griswold, M.T. *The Icon Programming Language*, Third Edition, Peer-to-Peer Communications, 1996.
- [8] Hanus, M. The integration of functions into logic programming: From theory to practice. *J. Logic Programming*, 1994, 19&20:583-628.
- [9] McNamara, B., and Smaragdakis, Y. FC++: Functional programming in C++. In *Proc. Int'l Conf. Functional Programming (ICFP)*, Montreal Canada, Sept. 2000.
- [10] McNamara, B., and Smaragdakis, Y. Functional programming in C++ using the FC++ library. *ACM SIGPLAN Notices*, April 2001, 36(4):25-30.
- [11] Müller, M., Müller, T., and Van Roy, P. Multi-paradigm programming in Oz. In *Visions for the Future of Logic Programming: Laying the Foundations for a Modern Successor of Prolog*, Dec 1995.
- [12] Smolka, G. The Oz programming model. In *Computer Science Today*, Springer-Verlag, 1995.