Blending the Logic Paradigm into C++

Roshan Naik
roshan@mpprogramming.com
Last updated: Feb 25th, 2008

Revision History:
April 11th 2007: First version
May 8th 2007: In section 4.1, assignment of plain functions to type relation no longer allowed. Minor aesthetic updates.
Feb 25th 2008: Revised (with simpler) implementations for relational &&, || and eq() in sections 4.3, 4.4 and 4.5.

Abstract

The Logic paradigm (LP) provides a powerful programming model that has been underrepresented in mainstream programming as compared to the object-oriented, functional and imperative paradigms. Lot of work can be cited in the space of integrating logic into functional programming or the functional into imperative paradigm but relatively less has been written about integrating logic programming into popular object-oriented languages used by the majority of software developers. Consequently a vast unexplored territory exists in the space of multiparadigm programming techniques that leverage logic programming in the dominant universe of imperative and object–oriented programming.

This paper introduces a technique for seamlessly integrating Logic programming techniques into C++. In this approach, support for LP is provided in the form of a few library primitives and does not require any extensions or modifications to the C++ language. The proposed technique is directly based on the imperative paradigm and can be implemented in about 400 lines of C++ code. The code presented here is based on the Castor library. More advanced facilities, built using the core parts described here, are provided in Castor but not covered in this paper.

Categories and Subject Descriptors D.1.6 **[Programming Techniques]**: Logic Programming; D.1.5 **[Programming Techniques]**: Object-oriented Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques – software libraries*;* D.3.2 [**Programming Languages**]: Multiparadigm languages.

General Terms Design, Languages.

Keywords Multiparadigm Programming; Logic Programming; Object-oriented; Programming Languages; C++; Prolog

1. Introduction

Although most programming languages have one programming paradigm at their core, they often tend to integrate features from other paradigms. The degree to which they integrate aspects of other paradigms varies from language to language. For example functional and logic programming languages typically tend to rely on imperative features for performing I/O. Similarly imperative paradigm based languages like C++, Ruby, Python and more recently C#, have integrated some functional features.

The Logic paradigm has seen relatively less exposure in mainstream languages. Its highly declarative nature that focuses more on problem specification rather than how to solve a problem is a departure from most other paradigms. Evidence of its usefulness in conjunction with other paradigms has been well established in academic and non-academic research efforts. For example the multiparadigm techniques demonstrated in [1] provide an excellent glimpse into the many new possibilities brought about by integration of LP into an object-oriented and functional framework.

An obvious approach to supporting LP in C++ is to embed a Prolog style rule interpreter in the form of library. Having to explicitly interact with such an LP interpreter via API calls makes the final multiparadigm solution somewhat unpleasant. Observing efforts that use this approach, such as LC++ [7], reveals that LP code visibly stands out from regular C++ code as the original flavor of C++ is lost. Seamless integration of LP with other paradigms is important for having a clean programming model which is essential to promote the use of multiparadigm techniques.

The technique described in this paper allows relations to be defined as templates, classes, functions and even simply expressions. This low level of integration reduces the syntactic overhead and allows programmers to freely mix LP with the other paradigms available in C++.

We begin in section 2 with a look at some examples to get a feel for LP in C++ and also observe similarities with Prolog. Section 3 discusses how relations are executed in C++ and finally section 4 covers the essential implementation details involved in supporting LP model in C++. Basic familiarity with Prolog or logic programming is expected.

2. Logic programming in C++

The following Prolog example lists facts about a particular family from Greek mythology.

```
male(zeus).   /* “zeus” is a male */
male(castor). /* “castor” is a male */

female(leda).
female(clytaemnestra).

/*zeus and leda are castor’s parents*/
parents(zeus, leda, castor).
parents(zeus, leda, clytaemnestra).
```

In C++, the equivalent code can be written using Castor as follows.

```
relation male(lref<string> x) {
  return   eq(x,"castor")
        || eq(x,"zeus");
}

relation female(lref<string> x) {
  return   eq(x, "leda")
        || eq(x, "clytaemnestra");
}


relation parents(lref<string> father
                 , lref<string> mother
                 , lref<string> child) {
 return eq(father,"zeus") && eq(mother,"leda")
                  && eq(child,"castor")
     || eq(father,"zeus") && eq(mother,"leda")
                 && eq(child,"clytaemnestra");
}
```

A few things are immediately noticeable in the C++ code. Each Prolog relation has a corresponding function with the same name. Such C++ functions are also referred to as relations. The return type of relations is `relation` and the parameter types are specified in terms of template type `lref`. Template type `lref` (abbreviation for *logic reference*) provides a facility for passing values in/out of functions in the form of arguments, similar to references in C++. Logic references, unlike C++ references, do not have to be initialized. The template arguments to `lref` describe the actual underlying type of the function’s parameter. In this example, all relations operate on strings. Under the hood, `lref` is essentially a reference counted smart pointer. The return statement in each relation is an expression composed of operators `&&`, `||` and calls to function `eq`. Function `eq` provides support for unification. That is, its job is to try to make both arguments equal. If its one of its argument is not initialized, it will be assigned the value of the other argument. If both arguments are initialized the two are compared using operator `==`. At least one argument to `eq` must be initialized. Unification fails only if comparison returns false. This unification strategy is a variation to that supported in Prolog and will be discussed further in section 4.3.

Function `eq` does not actually perform the unification immediately when invoked; instead it returns a function object that will, when executed. Similarly the `&&` and `||` operators return function objects which perform the conjunction and disjunction operations when executed. Thus, the return statements in the relations above merely return function objects and perform no real evaluation. These function objects encapsulate the semantics expected from the expression in the respective return statements. The return type `relation` essentially serves as a mechanism for holding such function objects.

The following Prolog rule describes the sibling relationship:

```
/* X & Y are siblings if parents of X are same
as parents of Y, and X is not equal to Y */
siblings(X, Y):-
      parents(X,M,F), parents(Y,M,F), X=\=Y.
```

In C++ we write the same rule as:

```
relation siblings( lref<string> x
                 , lref<string> y)  {
   lref<string> f, m;
   return parents(f, m, x)
     && parents(f, m, y) && predicate(x!=y);
}
```

Here the call `predicate(x!=y)` is interesting to note. Relation `predicate` is provided by Castor for easily converting functions/function objects that return `bool` into relations. In the above code, the expression `x!=y` creates a function object that tests `x` and `y` for inequality. This function object is then used as argument to `predicate`. Expressions involving logic references and standard overloadable operators (other than `*`, `&`, `->`, `[]`, `()` and `=`) are used to create simple anonymous function objects directly inline. These expressions are called *Inline Logic reference Expressions* or ILE. ILEs provide a convenient way to create simple anonymous function objects declaratively. This is somewhat similar to lambda functions facility available in functional languages. Due to the use of logic references, ILEs also automatically get an efficient support for closures with both lvalue and rvalue semantics[1]. A closure is the set of lrefs and values needed for evaluating the ILE. For the duration of its lifetime, every ILE has access to its closure even after the termination of the scope where the ILE and its lrefs were created.

ILEs are also useful in creating simple anonymous relations directly inline. For this we pass an ILE that produces a `bool` to relation `predicate`.[2] Since ILEs are primarily a convenience feature and not critical to implementing the

[1] Lrefs exhibit lvalue semantics and all other objects and values in the ILE expression exhibit rvalue semantics. Thus updating the rlef inside an ILE causes the original object to be modified.

[2] Early on, support for creating relations directly out of a boolean ILE expressions without need for an “adapter” relation like `predicate` was implemented, but this facility was later withdrawn as it ran into several problems including some limitations imposed by C++. The current design still leaves the door open to revisit this facility in the future (perhaps in C++0x) with an eye towards preserving compatibility.

logic paradigm, a discussion of their implementation and other interesting uses is beyond the scope of this paper.

Given the above relations, in Prolog we can ask the question "Is Castor a sibling of Clytaemnestra?" as follows:

```
?- siblings(castor, clytaemnestra).
```

And in C++:

```
if( siblings("castor","clytaemnestra")() )
  cout << "yes";
```

The application of the function call operator on the function object return by `siblings` triggers the search for an answer. In the above example we posed a question to check whether a sibling relationship exists between Castor and Clytaemnestra. Another type of question that can be asked is "Who is a sibling of Castor?". In Prolog this is phrased as:

```
?- siblings(castor, X).
```

Here `X` is an uninitialized variable for which we are interested in seeking a value. In C++ we can pose the same question as follows:

```
lref<string> sib;
siblings("castor", sib)();
cout << *sib << " is castor's sibling";
```

Since logic reference `sib` is not explicitly initialized to any string value, the backtracking and unification process will initialize it with a value that satisfies the question.

Questions such as the above can have multiple satisfactory solutions. Iterating over all solutions in C++ is quite naturally supported using the imperative looping constructs `while` and `for`. The following code prints all sibling pairs:

```
lref<string> sib1, sib2;
relation allSibings = siblings(sib1,sib2);
while( allSiblings() )
  cout << *sib1 << "," << *sib2<< "\n";
```

3. Execution of relations

In this section we take a brief look at how evaluation of relations occurs when triggered by application of the functional call operator. Consider the following relation describing spouse relationships written directly inline as an expression.

```
lref<string> h, w;
relation spouse =
           eq(h,"husband1") && eq(w,"wife1")
        || eq(h,"husband2") && eq(w,"wife2");
```

Variable `spouse` now holds a function object representing the expression tree shown in Figure 1.

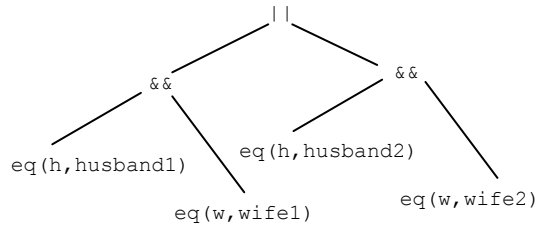

Figure 1. Expression tree inside relation spouse

In order to test if husband2 and wife2 are spouses we initialize `h` to `"husband2"` and `w` to `"wife2"` and trigger the evaluation of the relational expression by applying the function call operator on spouse:

```
h="husband2";  w="wife2";
if( spouse() )
    cout << "Yes";
```

The evaluation of `spouse` relation proceeds as follows:

- `eq(h,"husband1")` is evaluated with `h` initialized to `"husband2"`. Since both arguments are initialized, this results in comparison of the two arguments. String comparison of `"husband2"` with `"husband1"` fails, which implies the failure of the entire `&&` expression to the left of the `||` operator. Short circuit evaluation allows us to skip evaluation of `eq(w,"wife2")`.
- Execution now proceeds to the right side of the `||` operator. Evaluation of `eq(h, "husband2")` succeeds as string comparison of `"husband2"` with `"husband2"` succeeds.
- Success of the left half implies that the right half must also be tried to determine the result of this `&&` expression. Evaluation of `eq(w,"wife2")` succeeds as `w`, which is initialized to `"wife2"`, is compared with `"wife2"`. The `&&` expression succeeds as both its halves have succeeded.
- Since one half of the `||` expression has succeeded, the evaluation of the entire `||` expression is considered a success and `true` is returned as the result for applying the function call operator to spouse.

Lets consider the question "Who is the spouse of husband1?". Our aim is to let backtracking and unification find a suitable value for `w` when `h` is `"husband1"`. For this we assign `"husband1"` to `h` and clear any value stored in `w` from the previous search.

```
h="husband1";
w.reset();
if( spouse() )
     cout << *w;
```

Evaluation proceeds as follows:

- `eq(h,"husband1")` succeeds as we have initialized `h` to `"husband1"`.
- Evaluation proceeds to the second half of the `&&` expression which is `eq(w,"wife1")`. Since `w` is not initialized

to any value, `eq` assigns its second argument `"wife1"` to `w` and returns `true`.
- The evaluation of the `&&` expression has thus succeeded and consequently the `||` expression too. Thus, true is returned to the `if` statement and the value stored inside `w` is printed.

Now let us consider what happens when `spouse()` is invoked for the second time:

```
if( spouse() ) cout << "2nd spouse:" << *w;
```

The goal here is obviously to see if another spouse exists for husband1. Relation `spouse` will resume evaluating its expression tree from the point where execution halted last time around. Before stepping through this execution we need to discuss a few things.

The expression tree contained in `spouse` is simply an aggregation of other smaller relations. Each node of the expression tree is a relation. Operators `||`, `&&` and function `eq` are all relations. Any relation in logic programming is considered to be capable of producing zero or more solutions. Only way to see how many solutions can be produced is to actually go through the process of generating all solutions. Each application of the function call operator on a relation triggers the search for the next solution. After all solutions have been discovered, `false` is returned to the caller and all future attempts to seek solutions on that relation will also fail and return `false`. Every time a solution is found, the uninitialized logic reference part of the relation (in this case `w`) will be assigned a value representing the solution. However, before proceeding to find the next solution, it is important that such side effects be reverted. Thus it is the responsibility of each relation (at any level in the expression tree) to revert any side effects that it has induced so far, prior to moving on to another solution.

Let us return back to the discussion of what happens when `spouse()` is invoked the second time.

- Execution resumes at `eq(w,"wife1")`. Essentially `eq` is being given a chance by the `&&` operator to either produce more solutions or report failure. `eq` takes this chance to revert the unification of `w` with `"wife1"` and then returns false to indicate there are no more solutions. Logic reference `w` has now been reverted to its original uninitialized state.
- Left half of the `&&` expression had previously reported success, which means there may still be more solutions coming from it. Thus evaluation returns to back to `eq(h,"husband1")`. Since `eq` has only performed the comparison of `h` with `"husband1"` there is no other task left for it to perform or any side effects to be revert. It immediately returns false to the `&&` expression.
- Failure of the left half of the `&&` implies that there is no need to purse the right half any more and false is returned back to the `||` operator.
- The `||` operator detects failure of its left half and proceeds to try out the right half. Evaluation reaches `eq(h,"husband2")`. Since `h` is initialized to a value, `eq` will perform comparison instead of assignment. Comparison of `"husband1"` with `"husband2"` results in failure and the result cascades up to the `||` operator and finally back to the if statement.

We can continue to safely invoke the function call operator on spouse. But henceforth, operator `||` will immediately return false as it remembers the fact that both its halves have been evaluated.

4. Implementing support for LP

Operator `&&`, operator `||`, function `eq`, template type `lref` and type `relation` are the five key elements that provide the foundation for logic programming. Backtracking and unification are the two primary pillars of the computational model in LP. Backtracking support comes from overloads for operators `&&` and `||`. Function `eq()` along with help from template type `lref` provides support for unification. Template `lref` also provides a channel for bidirectional flow of information via functions parameters, similar to the pass-by-reference mechanism in C++. Finally, type `relation` is instrumental in enabling a clean programming model that requires minimal syntax when defining and consuming relations. The following sections cover these five elements in more detail.

4.1 Type relation

The type `relation` internally represents a function object with no arguments and return type `bool`. The simple approximate definition for type `relation` is in terms of `boost::function` from the Boost library [9]:

```
typedef boost::function<bool(void)> relation;
```

Figure 2, below, shows an alternate but equivalent definition that does not rely on the Boost library.

```
class relation {
  struct impl {
    virtual ~impl(){}
    virtual impl* clone() const=0;
    virtual bool operator()(void)=0;
  };

  template<class F>
  struct wrapper : public impl {
    explicit wrapper(const F& f_): f(f_)
    { }
    virtual impl* clone() const {
        return new wrapper<F>(this->f);
    }
    virtual bool operator()(void) {
        return this->f();
    }
  private:
    F f;
  };
```

```
  std::auto_ptr<impl> pimpl;

public:
  typedef bool result_type;

  template<class F>
  relation(F f)
         : pimpl(new wrapper<F>(f))
  { }

  relation(relation const& rhs)
         : pimpl(rhs.pimpl->clone())
  { }

  relation& operator=(const relation& rhs) {
    this->pimpl.reset(
              rhs.pimpl->clone() );
    return *this;
  }

  bool operator()(void) const {
      return (*this->pimpl)();
  }
};
```

Figure 2. Alternate definition for type relation.

One key difference between, `relation` and `boost::function<bool(void)>` is that only function objects, and not functions, can be assigned to `relation`.

"Type erasure" is the primary feature provided by `relation`. Type erasure allows objects of arbitrary types to be assigned to a `relation` object as long as they support the function call operator that returns a `bool` and takes no arguments. Once the assignment to a `relation` has taken place, the actual type of the object to the right hand side of the assignment is not retained. In other words, the actual type information is erased. The only type information left is the fact that it supports the function call operator that returns bool and takes no arguments[3]

To see the benefit of type erasure, consider the relation `male` from section 2. The actual type of the return statement is something like `Or<relation, relation>`. That is a slightly long type name for such a simple return statement. Breaking the type name down into smaller pieces will reveal its correspondence to nature of the expression in the return statement. The `male` relation could be alternatively been specified using this type as the return type instead of `relation`. However expecting the programmer to mentally compute such type names accurately is not reasonable and especially so when the return statements get more complex. Declaring the return type simply as `relation` instead, greatly reduces the burden on programmers and makes the final code more readable. Similarly, consumers of relations also benefit from ignoring the actual type name. Thus type `relation` plays a critical role in simplification of the overall syntax. A downside of choosing to ignore the exact type is a slight performance penalty incurred when redirecting the function call from type `relation` to the actual underlying type. For discussion on performance of type erased function calls refer to [9].

Type erasure also manages to contain the problem of "explosion of types" which can sometimes get severe. If types were preserved for each sub expression then the function object at every level of the expression tree has a unique type. And pervasive use of such "type preserved" relational expressions only leads to more and more types which will eventually break the compiler or linker. The downside however is a loss of performance since type erased function objects are poor candidates for inlining.

Type erasure is a pure library technique in C++ that relies on templates. More discussion on type erasure and its implementation aspects in C++ can be found in [13]. The following example demonstrates the essential semantics of `relation`:

```
bool alwaysTrue() {return true;}

struct neverTrue {
    bool operator() (void) {return false;}
};

relation t = &alwaysTrue; //Compiler Error!
relation f = neverTrue();

f();   // invokes neverTrue::operator()

relation nt = f;  // make a copy
nt();    // now invokes neverTrue::operator()

relation r; //ERROR! r must be initialized
```

In short, `relation` supports application of `operator()` with no arguments and, construction and assignment from predicate function objects.

An obvious alternative to type erasure is to use inheritance and polymorphism. Type `relation` could be defined as base class that declares the function call operator as a pure virtual function:

```
struct relation2 {
  virtual bool operator()(void)=0;
};
```

Although this alternative is workable, it is fraught with problems. First, it requires all user defined relations to be defined as classes that inherit from this type. For example the `male` relation we covered earlier would look as follows:

```
struct Male : relation2 {
  ..type..  x_;
  Male(..type.. x) : x_(x)
  { }
  bool operator()(void) {
    return unify(x_,"castor")
```

[3] Some experts feel the usage of the term "type erasure" to indicate these exact semantics in C++ may be somewhat inaccurate. The related terms "type erasure" and "type reconstruction" seem to have their origins in type theory.

```
            || unify(x_,"zeus");
  }
};
```

Such syntactic requirements get cumbersome rather quickly. Furthermore, this leads to the need for using pointers (or references) and memory management hassles. Consider the usage of such relations:

```
//pointers needed to invoke polymorphic opera-
tor()
relation2* bothMales = new Male(x) && new
Male(y);
```

The lifetime and ownership of the objects allocated using `new` is no longer clear as `relation` objects typically tend to be used outside the lexical scope in which they were instantiated. Additionally, the burden of explicit deallocation is placed on users. A quick fix for this problem is to use smart pointers. However it only makes the syntax poorer and the end result looses simplicity of expression.

Another possible technique, not discussed here, is the use of "duck typing" as discussed in [8]. However it leads to the problem of long type names similar to that discussed above. Use of `boost::variant` and `boost::any` was also considered for representing type `relation`, but did not prove to be fruitful.

4.2 Template lref: The Logic reference

The template type `lref` is an abbreviation for *logic reference* and provides a facility for passing values in/out of functions in the form of arguments, similar to references in C++. Unlike C++ references, logic references do not have to be initialized. Member function `defined()` can be used to check for initialization as demonstrated in the following code:

```
lref<int> li1=1; // li1.defined()==true
lref<int> li2 ;  // li2.defined()==false
li2 = 2;         // now li2.defined()==true
li2.reset();     // now li2.defined()==false
```

Operator * and operator -> can be applied to access the value referenced by a logic reference. Semantics of initialization and assignment for logic references are important to understand since these semantics allow lrefs to be used as in/out parameters:

- When an `lref<T>` is constructed or assigned with an object of type `T` (or a type convertible to `T`) or `lref<T>`, it internally stores a copy of the value. Thus an object of type `T` can be supplied anywhere an `lref<T>` is required.
- When an `lref` is copy constructed with another `lref<T>`, both logic references will be joined together.
- `lref<T>` cannot be joined with `lref<T2>` if `T2` is not the same type as `T`.

Joined references are also referred to as co-references. Logic references that are joined together will refer to the same underlying value. Any change in the underlying value of one logic reference is observable by all logic references that are joined with it. Such a join cannot be broken. That is, if logic references *A* and *B* are joined together and *C* and *D* are joined together, then *C*'s join with *D* cannot be broken to in order to form a join with *A* and *B*. C will continue to be a part of the join for the duration of its lifetime. Logic references can only be joined together by copy construction only (and not by assignment). The join is automatically destroyed when the logic reference is destroyed. When the last joined reference is destroyed, it will deallocate the underlying value. Logic references are also safe to use with polymorphic types. An object of derived type may be assigned to an `lref` of a base type. Figure 3 demonstrates the semantics of `lref` in more detail.

```
// Assignment of polymorphic types
Derived d;
lref<Base> lb = d;

// Assignment of convertible types
lref<string> ls="castor";

// Assignment same or diff lref types
lref<int> li1=1, li2=2;
li2=li1;              // copies integer
lref<double> ld;
ld=li;   // copy as double

// Copy construction
lref<int> li3=li1;  //join with li1

lref<double> ld = li3; /* ERROR! Can't join
lrefs of different type even if referenced
types are convertible */

// Accessing underlying value
int i = *li1;
cout << ls->length();

// Updating joined logic references
lref<int> li4 = li1; // joined
li4 = 25;
cout << *li1;  // prints 25
```

Figure 3. Semantics of logic references

Understanding the basic semantics described above is generally sufficient to produce an equivalent implementation. Since an implementation of `lref` is relatively straightforward to produce using the semantics described above, the implementation details are left out from this paper.

4.3 Relation eq: The unification function

As discussed previously, function `eq` provides support for unification. Relation `eq` attempts to *unify* its first argument with the second. Both parameters types are `lref<T>`. The unification algorithm implemented by `eq` is as follows.

- If both arguments are initialized, then they are compared for equality. Outcome of the comparison is returned as result.
- If only one of the arguments is initialized, its value is assigned to the other uninitialized argument and returns `true`.
- If both arguments are not initialized, an exception is thrown.

In short, unification is a "generate or compare" operation. If both arguments are initialized we compare them, otherwise we generate a value for the uninitialized one by assignment. Note that it is possible to implement other variations to unification, but is generally not required. This strategy is a slight deviation from that used in Prolog which allows uninitialized terms to be unified. Castor's approach guarantees that a logic reference will be definitely initialized when unification succeeds on it. This guarantee then cascades up to user defined relations. Consequently it is much easier (but not foolproof) to guarantee that logic references can be accessed safely without first testing them for initialization when backtracking and unification have reported success as follows:

```
lref<string> sib;
if( siblings("castor", sib)() ) {
  if( sib.defined() ) // this check NOT needed
    cout << *sib << " is castor's sibling";
}
```

The following pseudocode demonstrates these operational semantics using coroutine style implementation (by borrowing the `yield` keyword from C#). Here `lhs` and `rhs` refer to the two lref arguments to `eq`.

```
if (lhs.defined() && rhs.defined()) {
  yield return *lhs == *rhs; // compare
}
else if(lhs.defined()) {
  rhs = lhs;       // generate in rhs
  yield return true;
  rhs.reset();
}
else {
  lhs = rhs;       // generate in lhs
  yield return true;
  lhs.reset();
}
return false;
```

Figure 3a. Coroutine pseudocode for Unification

Due to lack of direct support for coroutines in C++, we need to simulate coroutines using function objects. The equivalent C++ code for the pseudocode in is give below:

```
template<typename T>
class Unify {
  lref<T> lhs, rhs;
  int state;
public:
  Unify(lref<T> lhs, lref<T> rhs)
    : lhs(lhs), rhs(rhs), state(0)
  { }

  bool operator()(void) {
    switch(state) {
    case 0:
      if (lhs.defined() && rhs.defined()) {
        state=4;
        return *lhs == *rhs;
      }
      else if(lhs.defined()) {
        rhs = lhs;
        state = 1;
        return true;
    case 1:
         state = 3;
         rhs.reset();
       }
       else {
         lhs = rhs;
         state = 2;
         return true;
    case 2:
         lhs.reset();
       }
       state = 3;
    default: // case 3:
      return false;
    } // switch
  } // operator()
};


template<typename T>
relation eq(lref<T> l, lref<T> r) {
    return Unify<T>(l,r);
}
```

Figure 3b. Code for Unification equivalent to Fig 3a

Function `eq` itself has a trivial one liner body that returns a function object of type `Unify<T>`. It is the job of this function object to perform unification sometime in the future i.e. when its `operator()` is invoked. The two arguments to `eq` are stored in `Unify` as data members `lhs` and `rhs`.

On first application of `Unify::operator()`, if both arguments have defined values, the two arguments are compared and the outcome of the comparison is returned. All future applications of `Unify::operator()` will return `false`. Unification merely returns `false`. On the other hand if one of the two arguments does not have a defined value, the value of the other argument is assigned to it and `true` is returned. On next application of `Unify::operator()`, the argument to which the value was assigned is reset back to its uninitialized state by invoking `reset()` on the lref and `false` is returned. All future applications of `Unify::operator()` return `false` right away.

Method `lref::reset()` is a trivial operation that merely clears an internal flag. There is no memory deallocation or destructor invocation is involved. Thus reverting state during backtracking is very efficient.

In section 4.1 above we discussed how relations are responsible for reverting any previously induced side effects before proceeding to find more solutions. Similarly, if assignment has been performed to `l`, the next time `Unify::operator()` is invoked, it needs to revert this assignment. Members `lchanged` and `rchanged` are used for tracking whether `l` or `r` was assigned a value. Assignment is undone by invoking `reset()` on the logic reference.

The function objects returned by the various calls to `eq` within, for instance, the `gender` relation are stitched together into bigger and bigger compound function objects using the `||` and `&&` operators. The `gender` relation finally returns one function object that encapsulates the entire expression. This is basically a declarative mechanism to create function objects on demand specifically for the relational expression at hand. Since the return type of user defined relations is typically `relation`, these functions objects are stored in a `relation` object and will be subject to evaluation in the future only when needed. Often when evaluating such compound expressions, some of its sub expressions may not require evaluation, and indeed those will not be evaluated. Short-circuit evaluation of operators `||` and `&&` is a simple example. This technique is an example of lazy evaluation.

An alternative design for relation `eq` is to overload the `operator==` instead. Thus relation `male` from section 2 could then be rewritten as follows :

```
relation male(lref<string> x) {
    return (x=="castor") || (x=="zeus");
}
```

However this design was rejected for a few reasons. The first reason is a philosophical one deriving from the observation that equality is not the same as unification. Unification is actually a combination of equality and assignment. A more technical reasoning comes from the observation that using a named function like `eq` frees up `operator==` for other uses. In Castor this freedom is effectively used to create ILEs. Another technical reason is that using a named function allows us to introduce other overloads for `eq` that may accept more than two arguments or different slightly semantics (e.g. `eq_f`). Also, using named function `eq` requires typing only one additional character, but is much safer since accidental omission of any brackets around the equality operators can lead to completely unexpected semantics in the above code.

Programmers with LP background will notice that unification semantics provided in Castor differ from traditional LP systems like Prolog. In these systems, unification is about creating a binding between variables rather than comparison and assignment. Prolog's semantics are quite useful and can be implemented in C++ but not supported in Castor. In Castor the only way of creating a binding between two lrefs is by copy construction. After the lref has been constructed, it is not possible to change its binding to another lref. In order to support dynamic change of bindings, the type lref needs to be modified to use an extra level of indirection to point to its data. The unfortunate consequence of this extra level of indirection is that it impacts the performance of all LP code even when it does not require such unification semantics. For this reason, Castor's unification semantics are more constrained. It is possible that in a future version of Castor these semantics could be supported if a way to address the performance issue is found.

The following sections discuss the overloads for operators `||` and `&&` which provide the backbone for the backtracking mechanism.

4.4 Operator || : Disjunction

Similar to function `eq`, operators `&&` and `||` are also relations implemented as coroutines. They are essentially higher-order relations that take two relations as arguments.

Responsibility for supporting backtracking primarily lies with operator `||`. Backtracking is simply about trying alternatives. The function object returned by operator `||` will evaluate the first (i.e. the left) argument to `||`. If this succeeds, `true` is returned. On the other hand, if evaluation fails, the alternative (i.e. the second argument) is evaluated. As any relation can possibly generate multiple solutions, operator `||` will switch over to evaluating the second relation only once all solutions from the first have been exhausted. Similarly the right argument will be given a chance to produce all solutions. Left and right relations will return `true` as long they have solutions. Once `rhs` has returned false operator `||` will return false back to the caller.

Semantics for operator `||` can be summarized as: *generate all solutions from the left side, then generate all solutions from the right side*. The following pseudocode demonstrates these operational semantics using coroutine style implementation (by borrowing the `yield` keyword from C#). Here `lhs` and `rhs` refer to the two argument relations to operator `||`:

```
while( lhs() )
  yield return true; //'yield' taken from C#
while( rhs() )
  yield return true;
return false;
```

Figure 4a. Coroutine pseudocode for Operator ||

The equivalent C++ code that simulates the coroutine is as follows:

```
class Or {
  int state;
  relation lhs, rhs;
```

```
public:
  Or(relation lhs, relation rhs)
                : lhs(lhs), rhs(rhs), state(0)
  { }

  bool operator()(void) {
    switch(state) {
    case 0:
      if(lhs())
        return true;
    case 1:
      state=1;
      if(rhs())
        return true;
    default:
      state=2;
      return false;
    } // switch
  }
};


Or operator || (relation lhs, relation rhs) {
  return Or(lhs, rhs);
}
```

Figure 4b. Code for Operator | | equivalent to Fig 4a

The arguments relations to operator || are stored in Or as data members lhs and rhs. Data member state is used for keeping track where the execution needs to be resumed on the next evaluation attempt.

4.5 Operator && : Conjunction

In order to understand the semantics of operator && consider the following code:

```
lref<int> a, b;
relation r =  (eq(a,1) || eq(a,2))
           && (eq(b,3) || eq(b,4));
while(r())
  cout << "(" << *a << "," << *b << ")";
```

This prints (1,3)(1,4)(2,3)(2,4) when executed. This example demonstrates the sequence of evaluations occurring on the sub expressions comprising the && relation.

Semantics for && can be summarized as: *all solutions from the right side of the && are produced for each solution from the left side*. The following pseudocode demonstrates these operational semantics using coroutine style implementation (by borrowing the yield keyword from C#). Here lhs and rhs refer to the two argument relations to operator &&:

```
relation tmp = rhs; //make copy of rhs
while( lhs() ) {
  while( rhs() )
    yield return true; //'yield' taken from C#
    rhs = tmp; // reset rhs
}
return false;
```

Figure 5a. Coroutine pseudocode for Operator &&

The equivalent C++ code that simulates the coroutine is as follows:

```
class And {
  int state;
  relation lhs;
  relation rhs, tmp;
public:
  And(relation lhs, relation rhs)
      : lhs(lhs), rhs(rhs), tmp(rhs), state(0)
  { }

  bool operator()(void) {
    switch (state) {
       case 0:
         while( lhs() ) {
    case 1:
           state=1;
           while( rhs() )
             return true;
        rhs = tmp;
           state=0;
         }// while
       default:
         state=2;
         return false;
    } // switch
  }
};



And operator && (relation lhs, relation rhs) {
  return And(lhs, rhs);
}
```

Figure 5b. Code for Operator && equivalent to Fig 5a

4.6 Other features

Other facilities such as support for unification of sequences, standard iterators, recursive relations, ILEs, exclusive-or operator, dynamic relations etc. are provided in Castor. Comprehensive support for cuts is also provided for finer grain control over the backtracking mechanism. It is worthwhile noting that this approach to LP, makes it is possible implement any relation using tradition imperative style code instead of the more natural relational style. Relation eq, and operators || and && are all examples of implementing relations imperatively. Sometimes this is essential to implement relations that bridge the object-oriented or other paradigms with relations. In other cases such techniques may be used to manually optimize relations that are performance critical. The downside of implementing relations in such a fashion is that they are tedious and error prone, and on the plus side they are easier to step through using a debugger. For further discussion on these topics refer to [1]

5. Related work

Prior efforts in the area of integrating logic and imperative paradigms have relied on many different techniques. Gen-

erally these approaches can be classified into those that require compiler support and those that are implemented as pure libraries. Modifying an existing language to make relations first class native concepts can be beneficial but is not always practical if the language has a well established user base. Providing language level support is sometimes done by embedding a Prolog style rule processing engine directly in the language as in Oz [5]. Leda [3] uses a different and very innovative approach by implementing support for LP partly in language and partly in library without incorporating an LP interpreter. Relatively minimal language support is provided for LP and rest of the support comes from just a few lines of library code. Leda's technique is sometimes referred to as the "continuation passing" approach and based on [12]. This strategy relies heavily on functional features such as lambda functions and closures. Application of a similar approach for supporting logic programming in Java (using a combination of language extensions and library) is discussed in [2] and [4].

LC++ [7] implements a Prolog style interpreter in library form to support LP in C++. Defining relations and finding solutions are then performed via APIs provided by LC++. The programming model leads to rather distinct boundary around the Logic paradigm since other paradigms have to interact with relational code via APIs. The syntax for LP also visibly stands out from regular C++ code. MPC++ [6] uses the continuation passing approach to provide LP support in pure library. MPC++'s approach blends user defined relations better with regular C++ code but suffers from syntactic overhead as it requires relations to be defined as classes. Also certain functional programming aspects surface in user code when attempting to iterate over solutions generated by relations.

6. Conclusion

This paper introduced a technique for integrating the logic programming paradigm into C++ without requiring language extensions. The approach is easy to implement, based on the imperative/object-oriented techniques and enables building comprehensive support for LP facilities in C++. Relations can be defined as templates, classes, functions, member functions or just expressions. This low level of integration allows LP to fit seamlessly into C++. In any multiparadigm framework it is essential to be able to freely mix different styles cleanly and with minimal syntactic overhead. This technique has been implemented successfully as part of the Castor library. Good performance is also essential for any programming technique to succeed in the large scale. In this paper we have made no mention about performance as work on measuring it is pending.

Although many interesting features that help logic programming and multiparadigm programming have been developed, the surface has been barely scratched. A plethora of multiparadigm programming techniques remain to be discovered. Ability to effectively use of a combination of programming styles that best serve the tasks is very useful. Terms such as "pure object-oriented" or "pure functional" are not necessarily very desirable features.

Acknowledgments

Timothy Budd's work on his multiparadigm language Leda served as inspiration for developing this technique. I am indebted to Tim Budd and Margaret Burnett at Oregon State University for introducing me to multiparadigm programming.

Thanks to Eric Niebler for explaining certain implementation details of his innovative regular expression library, Xpressive. This greatly helped crystallize the design of type `relation` which had previously proved troublesome.

Finally, without some of the innovative techniques developed and documented by numerous people in C++ community surrounding templates and metaprogramming this effort may not have been possible.

References

[1] Naik, Roshan., *Introduction to Logic Programming with C++*, 2006. Available at http://www.mpprogramming.com/resources/CastorTutorial.pdf.

[2] Naik, Roshan., *Multiparadigm Programming with Java/MP*. Masters Thesis, Oregon State University, 2001. Available at http://eecs.oregonstate.edu/library/?call=2003-2

[3] Budd, Timothy A., *Multiparadigm programming in Leda*. Addison-Wesley, 1995.

[4] Budd, T.A., *The Return of Jensen's Device.* In Multiparadigm Programming with Object-Oriented Languages, published by John von Neumann Institute for Computing, Jülich, Germany, 2002

[5] Van Roy, Peter, *Logic Programming in Oz with Mozart.* In Proceedings of the 1999 international conference on Logic programming (Las Cruces, New Mexico, United States).

[6] Edwards, Stephen H., *CS 5314 MPC++ Resources*. Available at http://courses.cs.vt.edu/~cs5314/Spring02/mpcpp.php

[7] McNamara, B., and Smaragdakis, Yannis. *Logic Programming in C++*. Available at http://www-static.cc.gatech.edu/~yannis/lc++/.

[8] Koenig, Andrew. *Templates and Duck Typing*. C/C++ Users Journal, June 2005.

[9] Gregor, Douglas. *Reference document for Boost.Function*. Available at http://www.boost.org/doc/html/function.html

[10] Friedman, Daniel P., William E. Byrd and Kiselyov, Oleg. *The Reasoned Schemer*. The MIT Press, 2005.

[11] Spivey, J.M. and Seres, S.. *Embedding Prolog in Haskell*. In proceedings of Haskell'99, Paris, France, 1999.

[12] Mellish and Hardy. *Integrating Prolog in the POPLOG Environment*. In Implementations of Prolog, J.A Campbell (ed.) Ellis Horwood, New York, 1984.

[13] Abrahams, D., and Gurtovoy, A.. *C++ Template Metaprogramming.* Addison Wesley, 2004