Logic Programming in C++ with the LC++ Library

Brian McNamara and Yannis Smaragdakis

College of Computing
Georgia Institute of Technology
lorgon,yannis@cc.gatech.edu

Abstract. This paper describes LC++, a library supporting logic programming in C++. LC++ preserves the expressive syntax of logic languages like Prolog but also adds features from C++ like static type-checking and arbitrary effects. Prior approaches to embedding a logic programming language into an existing object-oriented language lacked a sufficiently general interface for communicating the results of a logic query back to client OO code. In contrast, LC++ provides a concise, expressive interface between logic and OO code.

LC++ is built atop FC++, a library for functional programming in C++. Functional programming techniques, including lazy evaluation and higher-order functions, are the key to implementing an effective interface between logic code and OO code. The “impedence mismatch” between the control flow of logic programs and OO programs is bridged effectively by functional programming. Additionally, the FC++ library simplifies demonstrably the implementation of LC++.

LC++ is a complete language embedded in C++: LC++ has natural syntax and full static type-checking and semantic analysis, all using an unmodified C++ compiler. We discuss how this compile-time processing is made possible by C++ template meta-programming.

1 Introduction

Multi-paradigm programming languages have been a topic of research for decades. The basic appeal is clear: offering the programmer a choice of paradigms enables her to choose the one best suited to the problem domain. Recently there have been a number of good examples of languages and systems which combine functional and logic programming (e.g., [6,8,9,16,22]), as well as mature implementations which add functional features to object-oriented languages (e.g., [10,13, 18,21]), but there have been few examples that extend object-oriented languages with logic programming. This is perhaps because these two paradigms differ the most, creating an impedence mismatch at the interface between them.

We believe that an effective way to bridge the gap between object-oriented programming and logic programming is to use functional programming as a stepping stone. To that end, we present LC++. LC++ is a library for doing logic programming in C++. LC++ is built atop FC++[13,14], our library for doing
functional programming in C++. FC++ plays a key role in providing an elegant interface between LC++ and normal C++ code, and FC++ also makes the implementation of LC++ much simpler. Using functional programming techniques, we are able to provide a natural embedding of logic programming in an object-oriented programming language.

2 Example

Figure 1 shows an actual part of an LC++ program describing the Simpsons family relationships. The `assert()` function is used to add facts and implications to the database, and the `query()` function is used to run queries. The syntax for functors, values, and logic variables is very similar to that of Prolog; Table 1 summarizes the differences between LC++ syntax and Prolog syntax.

<table>
<thead>
<tr>
<th>Description</th>
<th>Prolog</th>
<th>LC++</th>
</tr>
</thead>
<tbody>
<tr>
<td>unification</td>
<td>=</td>
<td>==</td>
</tr>
<tr>
<td>conjunction</td>
<td>,</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>disjunction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>implication</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>not provable</td>
<td>not()</td>
<td>not_provable()</td>
</tr>
<tr>
<td>evaluation</td>
<td>is</td>
<td>is()</td>
</tr>
<tr>
<td>dummy logic var</td>
<td>_</td>
<td>_</td>
</tr>
</tbody>
</table>

Table 1. Syntax mapping between Prolog and LC++

LC++ uses the C++ type system and templates to enforce static typing of logic programming code. To declare a functor, we use the form `FUNn` (n denoting the arity of the functor) and specify the functor’s name as well as the types of its arguments. For example, `parent` in the preceding example is declared like this:

```cpp
FUN2(parent,string,string);
```

Similarly, logic variables must be declared in order to statically typecheck them. In our example, to declare logic variable `x` of type `int`, we say:

```cpp
DECLARE( x, int, 10 );
```

The surprising integer constant parameter will be explained in Section 4.2. As in Prolog, we use the convention of having a logic variable’s name begin with a capital letter.

Like Prolog, LC++ uses “is” to bind logic variables to results of a computation. In the Simpsons example, the `ancestor` relation computes the number of generations between direct relatives like so:
```c
FUN2(parent, string, string);
FUN2(father, string, string);
FUN2(mother, string, string);
FUN2(child, string, string);
FUN1(male, string);
FUN1(female, string);
FUN2(sibling, string, string);
FUN2(brother, string, string);
FUN2(sister, string, string);
FUN3(ancestor, string, string, int);

DECLARE(Mom, string, 0); DECLARE(Sib, string, 6);
DECLARE(Dad, string, 1); DECLARE(Sib2, string, 7);
DECLARE(Kid, string, 2); DECLARE(Anc, string, 8);
DECLARE(Par, string, 3); DECLARE(Tmp, string, 9);
DECLARE(Bro, string, 4); DECLARE(X, int, 10);
DECLARE(Sis, string, 5); DECLARE(Y, int, 11);

string bart="bart", lisa="lisa", maggie="maggie",
marge="marge", homer="homer", abraham="abraham";
lassert( male(bart) );
lassert( male(homer) );
lassert( male(abraham) );
lassert( female(lisa) );
lassert( female(maggie) );
lassert( female(marge) );

lassert( parent(marge,bart) );
lassert( parent(marge,lisa) );
lassert( parent(marge,maggie) );
lassert( parent(homer,bart) );
lassert( parent(homer,lisa) );
lassert( parent(homer,maggie) );
lassert( parent(abraham,homer) );

lassert( mother(Mom,Kid) == parent(Mom,Kid) && female(Mom) );
lassert( father(Dad,Kid) == parent(Dad,Kid) && male(Dad) );
lassert( child(Kid,Par) == parent(Par,Kid) );
lassert( sibling(Sib,Sib2) == father(Dad,Sib) && father(Dad,Sib2)
&& mother(Mom,Sib) && mother(Mom,Sib2)
&& not_provable( Sib==Sib2 ) );
lassert( brother(Bro,Sib) == sibling(Bro,Sib) && male(Bro) );
lassert( sister(Sis,Sib) == sibling(Sis,Sib) && female(Sis) );
lassert( ancestor(Par,Kid,1) == parent(Par,Kid) );
lassert( ancestor(Anc,Kid,X) == parent(Anc,Tmp) &&
ancestor(Tmp,Kid,Y) && X.is(plus,Y,1) );

query( father(Dad,Kid) );
query( sibling(maggie,Sib2) );
query( ancestor(Anc,bart,X) );
```

Fig. 1. Simpsons family relationships in LC++
lassert( ancestor(Par,Kid,1) == parent(Par,Kid) );
lassert( ancestor(Anc,Kid,1) == parent(Anc,Tmp) &&
        ancestor(Tmp,Kid,Y) && X.is(plus,Y,1) );

The code X.is(plus,Y,1) adds 1 to the current value of Y and binds the result-
ing value to X. plus is a functoid from the FC++ library[13,14], the functional
programming library that LC++ is built atop. FC++ functoids generalize the
notion of a function in C++; they can be bound to operators, functions, meth-
ods, or lambdas created on-the-fly. The general mechanism to do computation
in LC++ is

    SomeLogicVar.is( some_functoid, arg1, ..., argN )

Thus, via is(), LC++ has a general mechanism to “call out” to normal C++
code to perform computations or to have effects; just create a functoid that
describes the desired computation, and pass it as the first argument to is().

In the example above, lassert() was used to populate the database with
facts and implications, and query() was used to perform queries. But what does
query() do with its results? In fact there are three different query functions. We
shall wait until Section 4.1 to describe the main query() function. The simplest
of the other two is iquery(): it prints out its results. So if the final query from
the Simpsons example were

    iquery( ancestor(Anc,bart,X) );

then the results

Result #1
  - Anc = marge
  - X = 1

Result #2
  - Anc = homer
  - X = 1

Result #3
  - Anc = abraham
  - X = 2

would be printed to the screen.

The iquery() function returns an FC++ List of results. List is a lazy list
data type—the elements are not computed until actually requested. Each query
result is represented by an environment object (actually, a smart pointer (IRef)
to an environment object; more details to come in Section 4), which contains all
the information about the logic variable bindings. The type of the environment
object is a function of which (types of) logic variables appear in a particular
query. Naming Environment types can be difficult; the result type of the call

    iquery( ancestor(Anc,bart,X) );

is a List of

IRef<Environment<TL::CONS<K_TYPE,TL::CONS<Anc_TYPE,TL::NIL>>>
objects. That is, it is a lazy list of references to a binding environment holding
values for logic variables X and Anc. (By convention, the C++ type of logic
variable Foo is called Foo_TYPE; this type is declared by LC++ DECLARE macro.)

Such long and ugly type names are a common occurrence in C++ template
libraries, especially those using expression templates. Fortunately we can shield
the client from these names by providing a “type computer” which provides a
manageable alias for the type: the type QRT<Foo_TYPE,Bar_TYPE>::IE describes
the type of results of a query involving the logic variables Foo and Bar. (QRT
stands for “Query Return Type” and IE stands for IRef<Environment>.) As a
result, we can just say

typedef QRT<Anc_TYPE,X_TYPE>::IE IE;
List<IE> l = lquery( ancestor(Anc,bart,X) );

To get our list of results. We can then use the FC++ functions null(), head(),
and tail() to traverse the list of references to environment objects. The envi-
rонment object responds to the method at(SomeLogicVar), returning an object
representing the binding for that logic variable. Assuming the variable is bound,
the “*” operator returns the value it is bound to. This way the client can print
the results using her own choice of formatting. For example

while( !null(l) ) {
  IE env = head(l);
  std::cout << "X is " << *env->at(X) " and Anc is "
             << *env->at(Anc) << std::endl;
  l = tail(l);
}

will print to the screen:

X is 1 and Anc is marge
X is 1 and Anc is homer
X is 2 and Anc is abraham

FC++ Lists are lazy, and the implementation of the LC++ query functions
exploits this laziness in an important way. For instance, a query computation
may not terminate. The query may yield a few results and then get “stuck” in
an infinite loop, or the query may return an infinite number of results, such as
in this simple example involving the natural numbers:

FUN( nat, int );
DECLARE( X, int, 10 );
DECLARE( Y, int, 11 );
lassert( nat(0) );
lassert( nat(X) - nat(Y) && X.is(plus,Y,1) );

Running the query nat(X) produces an infinite list of results. This doesn’t
present a problem, though; if we are only interested in the first 3 results, we
just ask for those. That is, we can say

typedef QRT<X_TYPE>::IE IE;
List<IE> l = lquery( nat(X) );
for (int i=0; i<3; +i)
{
    IE env = head(i);
    std::cout << "X is " << *env->at(X) << std::endl;
    1 = tail(1);
}

and that code\(^1\) prints

X is 0
X is 1
X is 2

The implementation of this feature (laziness) is described at length in Section 4.1.

3 Why LC++ Is Different

There are some other projects which add support for logic programming to existing object-oriented languages. LC++ is unique compared to such work because it integrates cleanly the control flow of the imperative and the logic programming language constructs. The alternative approaches include SOUL (which extends Smalltalk), J/MP (which extends Java), and MPC++ (C++). We first describe some common aspects of those three, and then discuss details of each in turn. (Section 6 describes other related work.)

All three approaches (SOUL, J/MP, MPC++) suffer the same key drawback with respect to query results: they do not leave the client in control. In SOUL, the results of a query are returned as a Smalltalk OrderedCollection object; this means that examples that involve infinite objects, like nat from Section 2, cannot be realized. The problem is similar in J/MP and MPC++: the client passes in a block of code to be executed for each result produced by the query, and the query executes the client code on each and every result. In contrast, LC++ gives the client control of the query by returning the results as a lazy list; the client can demand a few results, continue on with some other computation, and demand more results later as needed. A second difference between these approaches and LC++ is that none of the other three approaches can duplicate the specialized semantic analyses that LC++ can do (to be described in Section 4.2).

SOUL, the Smalltalk Open Unification Language adds logic programming features to Smalltalk. The original SOUL system was just an interpreter; clients would specify assertions and queries as strings, using code like

```
rep := SOULRepository new.
rep assert: 'father(homer,bart). father(homer,lisa).'
results := SOULEvaluator eval: 'if father(?dad,?kid) in: rep.'
```

However new work[3] integrates SOUL into Smalltalk so that functors work like ordinary message-sends and Smalltalk objects can participate in unification.

---

\(^1\) Functional programmers would undoubtedly prefer to write the code with `take()` and `map()`: FC++ provides these and many more of the functions in the Haskell Standard Prelude.
creating a truer embedding. Like LC++, SOUL preserves the declarative syntax that languages like Prolog provide. SOUL provides no static guarantees, however, since Smalltalk is dynamically typed and the SOUL implementation works with the reflection facilities of Smalltalk.

The J/MP language[2] is a Java extension supporting multi-paradigm programming, including logic programming. Logic programming in J/MP is enabled by using the Relation class and pass-by-name parameters. The && and || operators are overloaded (as in LC++), and the method unify() performs unification. J/MP has a weak logic programming model; unification can only be performed on a variable with a value—two unbound logic variables cannot be unified. Also, J/MP defines relations using notation that is more operational than declarative; for example in J/MP one might write

```java
public static Relation father( +String Dad, +String Kid ) {
    return parent(Dad, Kid) && male(Dad);
}
```

to define the father() relation described in our original LC++ example.

The MPC++ library[7] adds support for logic programming in C++. Like LC++, MPC++ is build atop FC++[13,14]. MPC++ has been used in a graduate programming languages course to teach students about different programming paradigms. Perhaps as a result of this use-context, MPC++ exposes more implementation details to clients, resulting in verbose code. When declaring an MPC++ functor, all known facts about the functor need to be expressed in a closed definition. (This is also true of J/MP.) For example, male() would be defined like this in MPC++:

```java
class Male : public Logic_Rule {
    Logic_Variable<string> person;
    public:
        Male( const Logic_Variable<string>& p ) : person(p) {} 
        Logic_Relation Rule_Definition() {
            return (person != "bart") || (person !="homer")
                || (person != "abraham");
        }
};
```

Thus, MPC++ has a static point of definition of all facts pertaining to a functor, while LC++ allows more facts to be added with lassert() dynamically, based on the control flow of the program. A static approach is more amenable to optimizations, but MPC++ does not attempt to optimize queries in any way (nor does J/MP).

Finally, we note that the Logic_Rule and Logic_Relation classes of MPC++ serve a similar purpose as the Relation class in J/MP, and operators are overloaded (MPC++ uses |= for unification). Indeed, the implementation strategies of J/MP and MPC++ are quite similar.
4 Beneath the Surface

In this section we describe two of the most interesting aspects of the LC++ implementation. First, we discuss how LC++ implements its control flow, using FC++ lazy lists as a natural way to return query results one-at-a-time on demand. Second, we discuss the use of C++ "expression templates" to perform compile-time computation, enabling LC++ clients to use Prolog-like syntax but have the C++ compiler parse, type-check, and semantically analyze this code.

Unfortunately, the semantic gap between these two parts is great: the discussion of the LC++ control flow should be straightforward from a logic programming standpoint, but not from the standpoint of a C++ template programmer. The inverse is true of the discussion of the LC++ compile-time computations. In the following, we try to offer appropriate background when necessary in order to make both sections accessible to readers from either background.

4.1 Query execution and C++ interfacing

LC++ represents query results as an Environment object, which maps each logic variable to its binding information. Since this is C++, we use effects to obtain an efficient implementation. That is, we use destructive update to modify the environment; unification causes bindings to be added; backtracking causes bindings to be removed.

One common way to implement Prolog queries is to mimic continuation passing style (CPS). A continuation is a function which embodies "what to do next" (after the current action is completed). Using CPS makes it easy to implement Prolog's unusual control flow; any particular term can either continue forward through the query (by calling the next continuation when this portion of the query succeeds) or backtrack (by returning to its caller when this portion of the query fails). The activation stack holds the "undo" information used for backtracking, and the continuation parameter holds the "future" (the rest of the query to be evaluated). Other logic programming approaches, like MPC++[7] and J/MP[2], use explicit CPS to express the logical control flow in an imperative language.

The problem with this approach is that the action describing "what to do with the query results" must itself be passed into the query as the final continuation. In specific cases, this is not a problem: for example, if all you want to do is print out all the results, then it would be easy to create a continuation function which just prints the contents of the Environment, and to pass this continuation into query(). In the general case, however, a client of query() may want to use the results in some arbitrary way using arbitrary C++ code, and there is no general mechanism for creating a continuation out of "the rest of your C++ program".

Put another way, the problem is the impedance mismatch between normal C++ control flow (which uses an activation stack) and LC++ control flow (which is effectively CPS). In simple cases where we are prepared to process all of the results at once, we can treat the query() function as a "stop the
world" process, which uses CPS to run the query and process all the results (query() would not return until all of the results are processed). However in the general case, a client may not be prepared to process all the results at once; the client needs query() to return the results (lazily) in a data structure which can be processed later. FC++ lazy lists solve the problem; representing the results as a lazy list effectively enables arbitrary C++ code to call query() and then continue on its way, co-routining with the CPS query functionality whenever the next result is demanded by the client. FC++ plays a role with dual significance here: FC++ lazy lists provide a smooth way to create the interface to LC++ queries, and the FC++ library makes the implementation much easier.

We now describe the actual implementation. At the lowest level, the main query() function returns a std::pair (standard C++ 2-tuple) whose first element is a reference to an Environment object and whose second element is a List<Empty>. Empty is a “nothing” data type—a struct with no members. List<Empty> signifies that the lazy list does not contain real values—it is only useful because traversing it produces side-effects on the Environment.²

The purpose of the List is to give clients control over query evaluation. As each element of the List is demanded, the LC++ query runs to produce the next result by side-effecting the Environment object. When the List finally becomes NIL this means there are no more query results. Thus a client calls query() like this:

```cpp
typedef QRT<Anc_TYPE,X_TYPE>::IE IE;
std::pair<IE,List<Empty> > p = query( ancestor(Anc,bart,X) );
IE env = p.first;
List<Empty> l = p.second
while( !null(l) ) {
    std::cout << "X is " << *env->at(X) << " and Anc is " 
              << *env->at(Anc) << std::endl;
    l = tail(l);
}
```

The query() and iquery() functions (presented back in Section 2) are built atop this interface.

We now illustrate how query() is implemented with some code. A query is represented as a Term object. Terms can be conjunction terms, disjunction terms, unification terms, etc. All of them support a run method with the following interface:

```cpp
struct Term {
    virtual List<Empty> run( IRef<Term> future ) = 0;
};
```

CPS is evident; in order to run a portion of a query, we must pass the remainder of the query (the continuation) as the future parameter. (Recall that IRef<Term> is like a Term pointer; FC++ provides the IRef class as a reference-counted pointer.) The run() method in Term returns a List<Empty>.

The body of the query() function ends with this code:

² We use List<Empty> because we cannot create a List<void> for technical reasons.
// "t" is a reference to the current Term
// "env" is a reference to the current Environment
List<Empty> l = curry2( ptr_to_fun(&Term::run), t, end_of_query );
// line above is FC++ lazy version of:  l = t->run( end_of_query );
return std::make_pair( env, l );

end_of_query is just an instance of a Term whose run() body says

    return cons(Empty(),NIL); // cons() is the FC++ List constructor

In other words, when we reach the end of the query, we should indicate one
result by returning a one-element List.

Further examples help illuminate what is going on. Consider DisjunctImpls
(the “or” terms created with ||). Here is the implementation, which just uses
the FC++ cat() function to concatenate two lists:

struct DisjunctImpl : public Term {
  IRef<Term> lhs, rhs;
  List<Empty> run( IRef<Term> future ) {
    return cat( lhs->run(future),
               curry2(ptr_to_fun(&Term::run),rhs,future));
    // FC++ lazy version of "cat( lhs->run(future), rhs->run(future) )"
  }
};

and here is the code for conjuncts (&&):

struct ConjunctionImpl : public Term {
  IRef<Term> lhs, rhs;
  List<Empty> run( IRef<Term> future ) {
    IRef<Term> newfuture = new ConjunctionImpl( rhs, future );
    return lhs->run( newfuture );
  }
};

That is, given term1&&term2 and a future, we run term1 with term2&&future
as its future.

Finally, consider unification. LC++ values can be unified using the unify() function, which returns a result of type UnRes. This is a two-element structure:

struct UnRes {     // UnRes means "Unification Result"
  bool ok;
  Fun0 void undo;  // Fun0< void > is the FC++ type of a zero-argument
                   // functoid that returns no result; an "effect thunk"
};

If a unification fails, the ok field is set to false and the undo field is unused. If a
unification succeeds, the environment is side-effected with the new binding, the
ok field is set to true, and the undo field is set to a thunk which, when executed,
will remove the newly-created binding from the environment. This is important
to the run() method in UnificationImpl, which looks like this:
UnRes ur = lhs->unify(rhs);
if ( !ur.ok )
    return NIL;
else
    return cat( future->run( dummy_term ),
        before( ur.undo, const_(List<Empty>())) );
// The previous line creates an empty list which first calls "undo" thunk.
// (FC++: The const_() function "lazifies" a List value, whereas the
// before() combinator prepends a thunk to a function.)

The logic of UnificationImpl::run() reads as follows. First, try to unify the
left-hand-side with the right-hand-side. If this fails, return the empty list (there
are no results). Otherwise, the result is the catenation of (1) the results from
running the future (the rest of the query)\(^3\) and (2) an empty list with the undo
thunk prepended. This results in the effects happening at the right time. Since
unification succeeded, we have added a binding to the environment. We run the
rest of the query with that binding intact. After all of those results have been
processed (that is, when the client demands the next result after those results
created downstream from this portion of the query), we undo the binding created
by this unification (to effect backtracking).

### 4.2 Parsing and semantic analysis

In this subsection, we discuss the C++-specific implementation techniques that
LC++ uses to enable clients to express logic programs in C++ using the simple
declarative syntax of the library interface. Expression templates[23] are used
to parse LC++ rules and queries as C++ expressions, and template meta-
programming[4] techniques are used to do basic analysis of LC++ expressions
so that LC++ code works within C++'s static type system.

**Parsing and Representation** The syntax of LC++ is implemented by over-
loading the C++ language operators that appear in Table I. These overloaded
operators return values of types that reflect the syntax tree of the expression.
For instance, C++ operators like == and && are overloaded to create values of
type ImplicationRep and ConjunctRep, respectively. All the different “Rep”
types serve to represent different entities of the syntax tree. Logic variables
 correspond to C++ values of type LogicVariable<T>, where \( T \) is the type of the
logic variable (e.g., integer, string, etc.). For example, code like

\[
X \equiv 3 \quad \&\& \quad Y \equiv 4
\]

creates a value whose type is a tree with a ConjunctRep at the root, with two
UnificationReps below it, each of which has a LogicVariable<int> child and
an int child.

\(^3\) The dummy_term passed to future's run() method is a just a meaningless placeholder.
query() ensures that all Terms end with an end_of_query object, which never uses
its future parameter.
**Type-Checking and Semantic Checking** The C++ type system is Turing-complete, and C++ templates can be used for meta-programming in the type system. (The C++ template system is an untyped, pure functional programming language, where the atomic values are C++ types.) Using this feature of C++, we can perform arbitrary (but very cumbersome) computations at compile-time.

There are three main high-level results of the compile-time computation performed by LC++, listed here with an example of each:

- Type checking: ensuring that in the expression `x==1`, `x` is a logic variable of type `int` (and not, say, one of type `string`).
- General semantic checking: ensuring that a client cannot ask for `env->at(x)` from the result of a query not involving `x`.
- Specialized semantic checking: ensure that the named logic variables appearing in an `assert()` statement always appear in more than one location.

We achieve these results by using metaprogramming on “type lists” in the `Rep` classes; this is explained next.

Recall that client code to run a query looks like

```cpp
typedef QRT<Anc_TYPE,X_TYPE>::IE IE;
List<IE> l = query( ancestor(Anc,bart,x) );
```

The QRT type computer computes the type of an environment that has bindings for each of the logic variables named by its template parameters.

The complication is that `query()` must compute a value whose type is compatible with the type computed by QRT. To do this, the implementation of `query()` must (at compile-time) traverse the parse tree of the logic expression passed to it and compute the set of all logic variables that appear in the term.

The type representing this set should be the same as that computed by QRT. Discovering all of the logic variables used in a logic expression expression is done using the `Rep` classes. LC++ keeps for each `Rep` type (representing an LC++ program term) a list of all the logic variables that appear in the term. This compile-time list is maintained as a field of each `Rep` class called “LVs”. Rather than discussing here the details of manipulating compile-time lists of types in C++, we refer the interested reader to [4]. It suffices to accept as given the list primitives: `TL::NIL`, `TL::CONS`, and `TL::AppendList`. (The namespace `TL` stands for “type list”.)

As an example, consider the definition of the `ConjunctRep` class (instances of which are created by the overloaded `&` operator):

```cpp
template <class LHS, class RHS> struct ConjunctRep : public HasLV {
    typedef typename TL::AppendList<typename LV<LHS>::::LVs,
        typename LV<RHS>::::LVs>::::Result LVs;

    ...
}
```

4 More precisely, for each of the logic variable types. The library interface is specifically designed to try to ensure that logic variables are declared in such a way that there is a one-to-one correspondence between logic variables and logic variable types. Thus the results of compile-time computations (types) can be meaningfully mapped back into the program (variables).
LHS lhs;
RHS rhs;
ConjunctRep( const LHS& 1, const RHS& r ) : lhs(1), rhs(r) {}
The unique integer constant that appears in the type (10 in the example above) provides a way to order the logic variable types. This enables us to create a canonical representation of a set of logic variables as a list: the canonical list always has the types appear in increasing order of their unique integer constants.

The canonicalization process also filters out duplicates, so that queries like

```cpp
lquery( ancestor(Anc,bart,X) && X==1 );
```

do not go mistakenly creating environments with type

```cpp
Environment<TL::CONS<Anc_TYPE,TL::CONS<X_TYPE, TL::CONS<X_TYPE,TL::NIL> >> >> // X mistakenly
```

The end result of the above computation is the general semantic checking performed by LC++. The type computers inside QRT, the `query()` functions, and the `Rep` classes all work to make the C++ type system ensure that LC++ code is statically checked. The type computers ensure that the environment types match, so that a client cannot ask for, e.g., `env->at(X)` from the result of a query not involving X. The type information maintained by LC++ lets the normal C++ type rules check more basic requirements, such as ensuring that in the expression `X==1`, X is a `LogicVariable<int>` (and not, say, a `LogicVariable<string>`).

In addition to basic type checking that QRT facilitates, LC++ supports more sophisticated analyses which are specific to the domain of logic programming. One example of such a specialized semantic analysis is detecting one-time-use variables in calls to `lassert()`. Suppose that when the client wrote the code for the family relationship example, instead of writing

```cpp
lassert( child(Kid,Par) == parent(Par,Kid) ); // correct
```

she accidentally wrote

```cpp
lassert( child(Kid,Par) == parent(Mom,Kid) ); // oops, used Mom
```

where `Mom` had inadvertently been used in place of `Par` on the right-hand side. The resulting code is legal and type checks, but it does not describe the intended `child()` relation.

This type of error is automatically statically detectable because it violates the rule that logic variables appearing in an `lassert()` statement should always appear in more than one location. A logic variable that is used only once can be unified with anything; if the client does actually intend to use a “don’t care” logic variable, they should do so explicitly using the special variable “_”. We use meta-programming to write an algorithm which analyzes `lassert()` calls and forces the compiler to emit a warning when one-time-use variables are detected. (Using meta-programs to statically analyze code and emit domain-specific compiler diagnostics is a technique that has been used by other recent C++ libraries[12, 15,19].)
5 Limitations and Future Work

The current implementation of LC++ has a few important limitations and omissions, which we intend to address in the near future. Additionally there are a few new directions we plan to investigate.

Omissions. The current LC++ implementation omits a few notable Prolog entities, like the cut operator and a retract() function. These will be implemented in the next version of the library.

Parametric polymorphism. One major restriction with LC++ is that functors are monomorphic. For example, we can define append() to work on lists of integers, but cannot use the same append() for lists of other types (e.g. string, double, etc.). We will also lift this restriction in the next version of the library, using some of the same techniques we used to implement the FC++ library.

Performance. LC++ allows clients to update the database of facts and implications with calls to assert() at any time—even during the execution of a query. This freedom effectively limits LC++ to executing queries like an interpreter, with all the associated performance limitations of that approach.

Static analyses and transforms. At the end of Section 4.2, we gave an example of one type of specialized semantic analysis that LC++ can perform at compile-time. Though we plan to add other types of semantic analyses to the library, we would also like to investigate if we can use static type information to enable some compile-time optimizations, such as re-structuring query trees.

Parameter modes. Some logic programming languages, such as Mercury[16] and HAL[5], enable the programmer to annotate functors with mode and determinism declarations. These declarations enable more static checking and can help provide better run-time performance. It would be interesting to extend LC++’s static type system so that parameter modes could be expressed.

6 Related work

Section 3 described other approaches to adding logic programming to existing object-oriented languages. Here we describe other work involving multiparadigm programming which includes a logic-programming component.

There are quite a few recent examples combining functional and logic programming. Gödel[9], Escher[8], Curry[6], and Toy[22] are a few examples of languages in this area. Each of these languages features static typing, polymorphism, and a pure (effect-free) style of programming. Mercury[16] fits into this group, but it also supports parameter mode and determinism declarations.

Unlike LC++, which combines logic and object-oriented programming by adding logic programming features to an existing OO language, some approaches start with a logic programming language and extend it with object-oriented features. One example in this area is Jinni[11]. Jinni is an interpreter for an extended version of Prolog (which includes features like classes and inheritance) that is written in Java. Jinni uses Java’s reflection capabilities to provide a
mechanism for the logic code to “call out” to Java, but the interface is heavy and there is a clear delineation between logic code and object-oriented code.

A few languages are designed to support all three (logic, functional, and OO) paradigms. The language Oz[17] combines all three paradigms in a dynamically typed, concurrent programming language. Leda[1] was specifically designed as a language for teaching the three paradigms; Leda is statically typed.

7 Conclusions

The LC++ library embeds a Prolog-like logic programming language inside C++. By utilizing the extensibility features of C++ (like operator overloading), we manage to preserve the declarative syntax and feel of logic programming. By using functional programming techniques (including laziness and combinators) we provide a natural interface to object-oriented C++ code. By utilizing C++’s powerful type system (through template metaprogramming), we are able to perform static semantic analyses at compile-time, including those specific to the domain of logic programming. Overall, LC++ creates a foundation to use C++ as an interesting new implementation platform for logic programming.

References

3. Brighan, Johan; Gybels, Kris; and Wuyt, Roel. “Towards Linguistic Symbiosis of an Object-Oriented and a Logic Programming Language”, Multiparadigm Programming with Object-Oriented Languages (MPOOL), Malaga, Spain, June 2002.
http://www.csse.monash.edu.au/~mbanda/hal/index.html
http://www.informatik.uni-kiel.de/~mh/curry/
http://courses.cs.vt.edu/~cs5314/Spring02/mpcpp.php
8. The Escher programming language 
http://www.cs.bris.ac.uk/~jw1/escher.html
9. The Gödel Programming Language 
http://www.cs.bris.ac.uk/~bowers/goedel.html
10. Järvi, Jaakko and Powell, Gary. The Boost Lambda Library. Available at 
http://www.binnetcorp.com/Jinni/
http://boost.org/libs/static_assert/static_assert.htm
15. McNamara, Brian and Smaragdakis, Yannis. “Static Interfaces in C++” *Workshop on C++ Template Programming* October 2000, Erfurt, Germany. Available at http://www.onumerics.org/tmp00/