Logic paradigm for C++

General purpose declarative programming

BoostCon

May 13th, 2010

Roshan Naik

roshan@mpprogramming.com

Key Terms

- Declarative Programming
  - Specify *WHAT* to compute
  - Not *HOW*
  - E.g. SQL, HTML (domain specific)

- Logic Paradigm.
  - General purpose declarative paradigm
  - Turing complete
  - E.g. Prolog, Gödel.

Agenda

- Introduction to Logic Programming (LP) concepts.
- Introduction to facilities available for LP in C++. Will use the open source library : **Castor** (www.mpprogramming.com).
  - Early stages of the process for inclusion into Boost. There appears to be interest.
- Plenty of code and a few “new” things.

Underlying Theme

- LP as a general purpose paradigm.
- Demonstrate Multiparadigm Programming (MP) by mixing LP with the Imperative, Object-oriented, Functional and Generic paradigms.

Declarative vs. Imperative

- Why are most programming languages imperative ?
- What is a key weakness of declarative programming ?
- What is a key weakness of imperative programming ?

Logic Paradigm (LP)

- **Computational model:** Predicate calculus (Turing complete).
- **Declarative :** Focuses on ***what*** to compute not ***how***.
- **Holy Grail of programming:** "*The user states the problem, the computer solves it*". LP is one approach in Computer Science.

- Basic mechanics of LP
  - Provide information to the computer.
    - Using relations
  - Computer employs a general purpose problem solving technique.
    - Consisting of backtracking and unification.

“relation”

- A set is a collection of (unique) objects.
- A simple way of thinking about relations is as an association/mapping between elements in two or more sets.

People	**Genders**	**GenderOf**
Frank	Male	(Frank, Male)
Sam	Female	(Sam, Male)
Mary		(Mary, Female)
Denise		(Denise, Female)

“relation”

- A relation is essentially a set.
- Thinking of relations as **mappings** between sets really helps when designing relations in LP.
- A relation is to the logic paradigm what a function is to the imperative paradigm.

Functions vs. Relations

- As functions

```
// check if (p,g)
bool checkGender(string p, string g) {
  ...
}
// get gender of p
string genderOf(string p) {
  ...
}
// get all people having gender g
list<string> havingGender(string g) {
  ...
}
// list all (p,g)
list<pair<string,string> > getItems() {
  ...
}
```

GenderOf
(Frank, Male)
(Sam, Male)
(Mary, Female)
(Denise, Female)

Functions vs. Relations

- As a relation

```
// declarative reading : p's gender is g
relation gender(lref<string> p, lref<string> g){
  return eq(p,"Frank") && eq(g,"male")
     ||  eq(p,"Sam")    && eq(g,"male")
     ||  eq(p,"Mary")   && eq(g,"female")
     ||  eq(p,"Denise") && eq(g,"female");
}
```

GenderOf

(Frank, Male)

(Sam, Male)

(Mary, Female)

(Denise, Female)

  - Specification is declarative.
  - One relation subsumes functionality of all four functions.

Demo

LP support

- Unification : **eq()**
- Operators : **&&**, **||**, **^**, **>>=**
- **ancestor(), parent(), …**

just ordinary relations

- Logic Reference : **lref<>**
- Type Erasure : **relation**
- Coroutines

Core

lref<T> : logic reference

- Reference counted smart pointer
- It does **not** have to be initialized.
- Dereferencing an uninitialized lref throws `InvalidDeref`
- Can enable/disable management of object:

```
int i=0;
lref<int> li (&i, false);  // will not manage i
li.set(new int(2), true);  // will manage obj
```

eq : The unification relation

- Semantically a combination of == and =

```
if (both args are initialized)
   return left == right;
else
   uninitializedArg = value of the other;
   return true;
```

- At least one of the two arguments must be initialized! Else throws `InvalidDeref`
  - Side Note: Prolog allows unification of uninitialized variables.

The magic type : relation

- Key to smooth integration of LP and simple syntax:
  - *Fairly* similar to `boost::function<bool()>`
    - Cannot assign functions (only function objects).
  - Type erasure at work

- Represents any function object that produces a bool and takes no arguments.

```
struct FuncObj {
    bool operator() (void) {...}
};
FuncObj f;
relation r = f;
r(); // execute FuncObj::operator()
```

The Operators

- Conjunction : operator **&&**

•Are themselves relations!
•Both arguments are relations

- Disjunction : operator **| |**
- Exclusive Disjunction : operator **^**
- Take Left : operator **>>=**

•Second arg is relation_tlr

  – ***Note****: ^ has higher precedence than || and && in C++*

Disjunction: Operator ||

- In plain English:
  - *Generate all solutions from 1st relation (one per invocation), then generate all solutions from 2nd relation (one per invocation)*.
- Coroutine style pseudo code

```
while( lhs() )
   yield return true;
while( rhs() )
   yield return true;
return false;
```

```
relation parent =father(..) || mother(..);

while( parent() ) {
   …
}
```

In C++ : Operator ||

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

```
struct Coroutine {
protected:
    int co_entry_pt;
};
```

```
while( lhs() )
    yield return true;
while( rhs() )
    yield return true;
return false;
```

```
template<typename L, typename R>
class Or : private Coroutine {
      L left;
      R right;
public:
      Or ( const L & lhs, const R & rhs) : left(lhs), right(rhs)
      { }

      bool operator() (void) {
         co_begin();
         while( left() )
               co_yield(true);
         while( right() )
               co_yield(true);
         co_end();
      }
};
```

Conjunction: Operator &&

- In plain English:
  - *Produce all solutions (one per invocation) in the 2nd relation for each solution in the 1st relation*.
- Coroutine style pseudo code (C# like syntax).

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

Exclusive Disjunction: Operator ^

- In plain English:
  - *Generate solutions from 2nd relation, only if 1st relation does not produce any*. (i.e. ExOr with short-circuit)
- Coroutine style pseudo code (C# like syntax).

```
bool lhsSucceded = false;
while( lhs() ) {
    lhsSucceded = true;
    yield return true;
}
while(!lhsSucceded && rhs())
    yield return true;
return false;
```

Take Left : Operator >>=

- .. Later

Examples

- Demo

Take Left : Operator >>=

- In plain English:
  - *Pass 1st relation as an argument to 2nd argument at the time of evaluation*.
- `operator >>=( relation lhs, relation_tlr rhs) {`

```
 while( rhs(lhs) )
  yield return true;
 return false;
}
```

e.g:

```
// items in sorted order
item(i,vec) >>= order(i);
// factorial of 5
range(n,1,5) >>= reduce(n, std::multiplies<int>());
```

TLRs
order
reverse
reduce
group_by
sum

group_by TLR

```
char firstChar(const string& s) { return s[0]; }
size_t str_len(const string& s) { return s.size(); }

lref<vector<string> > nums = //{"One","Two","Three".. }

// Single level grouping
lref<group<char,string> > g; // type of each group


lref<string> n;
relation r = item(n,nums) >>= group_by(n, &firstChar, g);

while(r()) {  // iterate over each group
    cout << "\n" << g->key<< ": ";
writeAll(g);
}
```

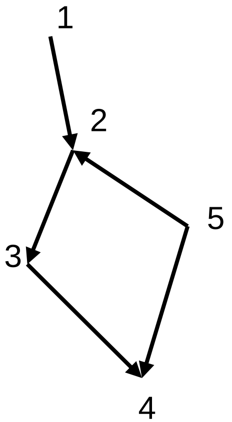
group_by TLR

```
// Nested grouping - two level
lref<group<char,group<size_t,string> > > g;

item(n,nums) >>= group_by(n, firstChar, g).then(str_len);

while(r()) {
   lref<group<size_t,string> > g2; // inner group
   relation subgroups = item(g2,g);
   while(subgroups()) {
       writeAll(g2)(); // print all items in subgroup
   }
}
```

Directed Acyclic Graphs




```
// Edges in the graph
relation edge(lref<int> n1, lref<int> n2) {
  return eq(n1,1) && eq(n2,2)
      || eq(n1,2) && eq(n2,3)
      || eq(n1,3) && eq(n2,4)
      || eq(n1,5) && eq(n2,4)
      || eq(n1,5) && eq(n2,2) ;
}

// Definition of path
relation path(lref<int> start, lref<int> end) {
  lref<int> nodeX;
  return edge(start, end)
      || edge(start, nodeX) && recurse(path,nodeX,end);
}
```

Binary Tree

```
class Tree {
    int value;
    Tree *left, *right;
public:
    relation item( lref<int> v ) const {
        // see next slide
    }
};
```

continued..

```
Tree::item( lref<int> v ) const { // C++0x
    long co_entry_pt=0;  // required by co_* macros below
    relation r = False();
    return [=]() mutable ->bool  {
        co_begin();
        if( left ) {
            for(r = left->item(v); r(); )
                co_yield(true);
        }
        for(r = eq(v,value); r(); )
            co_yield(true);
        if( right ) {
            for(r = right->item(v); r(); )
                co_yield(true);
        }
        co_end();
    };
}
```

Dynamic relations

- Following types provide support for dynamically building relations:
  - Conjunctions: &&
  - Disjunctions: ||
  - ExDisjunctions: ^
- These types are themselves relations.
- Any logic can be expressed statically, dynamically or as a combination of both.

Disjunctions : Dynamic relations

- Think of it as a dynamic list of clauses separated by || operator.
- `Disjunctions` is itself a relation.

```
vector<pair<int,int> > v = /* read edges from file  */;

relation edge(lref<int> n1, lref<int> n2) {
   Disjunctions clauses;
   for(... e = v.begin(); e!=v.end(); ++e )
      clauses.push_back( eq(n1,e->first) && eq(n2,e->second) );
   return clauses;
}
```

Examples

- Final Demo

Castor

- www.mpprogramming.com
- Pure header library (i.e. nothing to link).

- **Compilers supported by 1.1 beta**
  - GCC 4.4.1 and Visual C++ 2008
- **Compilers supported by v1.0:**
  - aCC, A.06.15 (Mar 2007)
  - C++ Builder 2007
  - GCC, v4.1.0
  - Visual C++ 2005 and 2008

Q / A

roshan@mpprogramming.com

Download:
www.mpprogramming.com/downloads/prebeta-1.1.zip

More information

- Tutorial
  - www.mpprogramming.com/downloads/betaTutorial.pdf
- Reference manual
  - www.mpprogramming.com/downloads/betaRefManual.pdf
- Design of Castor's core LP support
  - www.mpprogramming.com/downloads/betaDesignDoc.pdf