Introduction to Logic Programming in C++

Roshan Naik ([roshan@mpprogramming.com](mailto:roshan@mpprogramming.com))

**[DRAFT]** Last updated: Jul 27th, 2010

Version History

Feb 11th 2008: Initial version.

Jul 27th 2010 (Castor 1.1):

* Removed references to GenerativeRelation.
* Moved out section on “Implementing Relations imperatively” from section 2 into its own top level section 3.
* Rewrote section 3 on implementing relations imperatively.
* Added note about using pointers with lrefs in section 2.2.
* Added examples, other corrections.

TABLE OF CONTENTS

[Introduction to Logic Programming in C++ 1](#_Toc268447071)

[1 The Logic paradigm 3](#_Toc268447072)

[1.1. Facts 3](#_Toc268447073)

[1.2. Rules 4](#_Toc268447074)

[1.2.1. Recursive rules 4](#_Toc268447075)

[1.3. Queries and Assertions 4](#_Toc268447076)

[1.4. Computation by inferencing 4](#_Toc268447077)

[1.5. Summary 5](#_Toc268447078)

[2 Logic Programming in C++ 5](#_Toc268447079)

[2.1 Type relation 6](#_Toc268447080)

[2.2 lref: Logic reference 6](#_Toc268447081)

[2.3 Relation eq: The unification function 7](#_Toc268447082)

[2.4 Evaluating Queries 7](#_Toc268447083)

[2.5 Recursive Rules 8](#_Toc268447084)

[2.6 Dynamic Relations 9](#_Toc268447085)

[2.7 Inline Logic Reference Expressions (ILE) 10](#_Toc268447086)

[2.8 Sequences 11](#_Toc268447087)

[2.8.1 Generating Sequences 11](#_Toc268447088)

[2.8.2 Iterating over sequences 11](#_Toc268447089)

[2.8.3 Unification of Collections 13](#_Toc268447090)

[2.8.4 Summary 13](#_Toc268447091)

[2.9 Cuts – Pruning alternatives 13](#_Toc268447092)

[2.10 Relational Ex-Or operator 14](#_Toc268447093)

[2.11 Specifying Lref parameters types for relations 15](#_Toc268447094)

[2.12 Debugging 15](#_Toc268447095)

[3 Implementing relations imperatively 16](#_Toc268447096)

[With relation predicate 16](#_Toc268447097)

[With relation eval 17](#_Toc268447098)

[As coroutines 17](#_Toc268447099)

[4 Inline Logic Reference Expressions 18](#_Toc268447100)

[Creating relations from ILEs 19](#_Toc268447101)

[Limitations of ILEs 20](#_Toc268447102)

[Summary 20](#_Toc268447103)

[5 Limitations of Logic Paradigm 20](#_Toc268447104)

[Bi-directionality of lref arguments. 20](#_Toc268447105)

[I/O is not reversible. 21](#_Toc268447106)

[6 Logic Programming Examples 21](#_Toc268447107)

[Factorial 21](#_Toc268447108)

[Directed acyclic graphs. 21](#_Toc268447109)

[Finite Automata (FA) 22](#_Toc268447110)

[Query Expressions 24](#_Toc268447111)

[7 References 24](#_Toc268447112)

**Abstract:**

This paper is an introductory tutorial for Logic paradigm (LP) in C++. No prior experience is required with languages that natively support LP. It also demonstrates how LP blends with the other paradigms supported by C++ and also the STL. The ability to choose an appropriate paradigm or an appropriate mix of paradigms for solving a given problem is essential and at the heart of multi-paradigm programming. We begin with a brief introduction to the logic paradigm, followed by a discussion of logic style programming in C++ and finally conclude with examples. The primitives used here for logic programming are provided by Castor, an open source C++ library available from www.mpprogramming.com. No language extensions to C++ are required to compile the code provided here.

# The Logic paradigm

Logic programming is a Turing-complete programming paradigm. The model of computation used in logic is strikingly different from that of the more mainstream imperative and functional paradigms. Pure logic programs are entirely declarative in nature. When programming in languages based on the imperative paradigm (like C, C++, Java etc.), programmers actively instruct the computer *how* to solve a specific problem and the computer itself has no knowledge about the problem. Thus, algorithms play a central role. In logic programming languages such as Prolog or Gödel, however, it is exactly the opposite. In LP, the programmer provides problem-specific information to the computer instead of providing the steps required to solve a specific problem. The computer applies a general-purpose problem-solving algorithm to the domain-specific information to produce the desired results. The programmer is not involved with specifying the exact steps (i.e., the algorithm) used in solving the problem.

Information provided to the computer in logic programs can be classified into ***facts*** and ***rules***. This knowledge base of facts and rules describes the problem domain. Specific problems that we wish to solve in this domain are posed as questions or ***queries***. The computer examines the query in the context of the rules and facts and determines the solution. For example, if the game of Chess (or some other board game) represents our problem domain, the facts may consist of such things as:

* The different kinds of pieces (e.g., white pawns, black pawns, white king, etc.)
* The number of pieces of each kind (e.g., 8 black pawns, 1 white king, etc.)
* A description of the number of squares and their layout on the board (e.g., 8x8 board, 32 white squares, 32 black squares, etc.)

And the rules may consist of:

* The rule governing the movement of each kind of piece on the board (e.g., bishop moves diagonally)
* The rule to determine if a piece is under attack
* The rule to determine when a game is over and the result of the game.

Typical questions that arise in Chess (and many other board games):

* Given a specific layout of pieces on the board, what are all the possible moves for a given piece?
* Given a specific board layout, which pieces can be moved next?
* Which pieces are under attack in a given board layout?

Each question above represents a different but concrete problem that belongs to the problem domain of Chess. Shifting focus from describing how to solve a particular type of problem to describing the general rules of the broader problem domain allows us to seek answers to a wider variety of problems within the domain. In the remainder of this section we will further illustrate facts, rules and queries using a simple example concerning family relationships. The primary focus here is to get a feel for the basic mechanics of LP.

## Facts

Facts are essentially the simplest form of true statements pertaining to a problem domain. They may also be referred to as data. Let us consider a four -person family (Son: Sam, Daughter: Denise, Father: Frank, Mother: Mary, Grandparent: Gary. Here is one way of describing the facts pertaining to this family more accurately:

Children facts:

1. Sam is a child of Mary
2. Denise is a child of Mary
3. Sam is a child of Frank
4. Denise is a child of Frank
5. Frank is a child of Gary

Gender facts

1. Frank is male
2. Sam is male
3. Mary is female
4. Denise is female
5. Gary is male

The above facts can be used to answer simple questions such as “Is Sam male?”. This is a basic true/false question and can be answered by inspecting the gender facts. Since we have an exact match with fact 7, the answer to the question is “yes” or “true”. However, a similar question “Is Frank female?” yields “false” or “no”. This is because we do not have any gender fact stating Frank’s gender to be female. Thus, whenever matching evidence is found, the answer is true; and false otherwise.

A slightly different type of question is “What is the gender of Sam?”. This is not a true/false question but it still can be answered by looking up the gender fact 7. Another question could be “Who is the child of Frank?” Again this is not a true/false question, however, it is a little more interesting as there is not one but two answers to it, *Sam* and *Denise*. This tells us that we cannot simply stop examining the facts as soon as the first match is found; we need to continue the search till all relevant facts are exhausted. When there are multiple answers to a question, the person asking the question may be interested in one, some or all the answers.

A question that cannot be answered based solely on the above facts is “Is Mary a parent of Sam?” This is because we have not yet declared what it means to be a *parent*. There are no direct facts stating any parent-of relationships. To solve this we can add one fact of the nature “X is parent of Y” for each fact of the nature “Y is child of X” above. This approach is cumbersome and error-prone especially when the database of facts is large. A better approach is to use *rules* to infer these new facts.

## Rules

Rules are statements used to *infer* new facts (or data) from an existing body of facts. Here are some simple rules pertaining to the family relationships:

**Parent rule**: X is parent of Y if

* Y is child of X.

**Father rule**: X is father of Y if

* Gender of X is male **and** Y is child of X

**Mother rule**: X is mother of Y if

* Gender of X is female **and** X is parent of Y

Here X and Y are essentially parameters and not constants like “Sam” and “Frank”. The *parent* rule provides the ability to answer questions like “Is Mary a parent of Sam?” or “Who is a parent of Sam?”. Note that the *parent* and *father* rules above are specified only in terms of facts. The *mother* rule on the other hand is specified using a combination of facts (*gender*) and rules (*parent*) although it could be specified in same manner as the *father* rule.

### Recursive rules

Rules can also be specified in terms of themselves. Consider describing the ancestor relationship as a rule.

**Ancestor rule**: X is ancestor of Y if

* X is parent of Y **or**
* Z is parent of Y **and** X is ancestor of Z

Now we are in a position to ask “Is Gary an ancestor of Sam?” which should yield *true*. Similarly if we ask “Who is an ancestor of Sam?”, it should yield *Frank*, *Mary* and *Gary*.

## Queries and Assertions

Once we have built the knowledge-base consisting of facts and rules we are ready to ask questions. Specific problems that need to be solved are posed as queries. We have seen examples of queries above for the family relationships. Other examples of queries, for instance, when dealing with graphs is to ask “What is the shortest path between nodes A and B?” or “Is graph G a connected graph?”

Queries can be classified into two categories. The first kind simply tests if a certain statement is true: “Is Sam child of Gary?” These are also traditionally referred to as assertions, since the purpose is to essentially check or assert whether a fact is true. The second type of query is one that seeks for one or more solutions. For example, if we ask “Who is the child of Frank?”, we are not asserting if a fact is true, but instead asking the system to determine Frank’s child. We will henceforth refer to these as generative queries as it requires the generation of solutions.

## Computation by inferencing

We have not been very specific, so far, about how answers are actually computed (or inferred). Given the facts and rules described above it is fairly intuitive for any individual to mentally infer answers to all the questions we have asked. The fundamental principle behind such inferencing of new facts from existing rules and facts is referred to in formal Logic as *modus ponens:*

*If A is true and*

*A implies B,*

*then B is true.*

A and B, above, refer to arbitrary statements. This principle is at the heart of the computational model used in logic programming. An intuitive but rather rough approximation of the algorithm used to solve queries in logic programming systems such as Prolog is as follows:

1. Build a list of relevant facts and rules
2. Pick a relevant fact and see if it answers the question. Repeat this until all relevant facts are exhausted.
3. Pick a relevant rule that can be applied to derive new facts (using modus ponens). Repeat this until all relevant rules and facts are exhausted.

At any step of the inferencing algorithm, there may be more than one rule and/or fact that can be selected in order to continue execution. Each choice leads to a different inference path and not all paths necessarily lead to solutions. Paths that do not lead to solutions are abandoned and inferencing resumes from the most recent point where other facts and/or rules were available for inferencing. Abandoning the current path and resuming execution from an earlier point in order to make a different choice is known as *backtracking*. Backtracking continues till all paths of inference have been traversed. Thus, computation is reduced to traversing different paths of inference determined from the facts and rules available. This is similar to performing depth first search in a binary tree where leaf nodes represent candidate results and the inner nodes represent intermediate results.

In pure formal logic, the exact order in which facts and rules are selected for application is non-deterministic. This also implies that the order in which answers are obtained is not deterministic. Since some paths may lead to solutions quicker than others, in practice the order of execution is fixed (same as declaration order) to allow control over efficiency. Fixing the order of application of rules also simplifies reasoning about the execution of logic programs, which is important for debugging.

## Summary

In this section we described facts, rules, queries, assertions and inferencing. Facts are simply true statements. Rules can be used to derive new facts. Rules can be specified in terms of facts, other rules or even themselves (i.e. recursive rules). A collection of rules and facts can be used to answer relevant questions. Questions can be broadly classified into those that simply require a true/false answer or those for which solutions need to be generated. The former types of questions are referred to as assertions and the latter referred to as generative queries. Assertions have only one answer (i.e. true/false). Generative queries may have zero or more solutions. “Is Sam male?” is an example of an assertion. “Who is the child of Mary?” is an example of a generative query. Logical inferencing is used to answer questions. It involves examining the facts and application of rules. New facts emerge from application of rules which become candidates for future consideration during inferencing.

# Logic Programming in C++

Now let us translate the above facts and rules into C++ so that they can be executed. The examples here make use of Castor, an open source library that enables logic programming in C++. Castor enables embedding of logic style code naturally into C++ by allowing rules and facts to be declared as classes, functions or even just expressions. This low level of integration is very useful and allows for a programming platform where the paradigm-boundaries are seamless.

The following C++ functions represent the *child* and *gender* facts and the *father* rule (described previously) using Castor:

// c is child of p

**relation** child(**lref**<string> c, **lref** <string> p) {

return **eq**(c,”Sam”) && **eq**(p,”Mary”) //fact 1

|| **eq**(c,”Denise”) && **eq**(p,”Mary”) //fact 2

|| **eq**(c,”Sam”) && **eq**(p,”Frank”)//fact 3

|| **eq**(c,”Denise”) && **eq**(p,”Frank”)//fact 4

|| **eq**(c,”Frank”) && **eq**(p,”Gary”) //fact 5

;

}

// p’s gender is g

**relation** gender(**lref**<string> p, **lref**<string> g) {

return **eq**(p,”Frank”)&& **eq**(g,”male”) //fact 6

|| **eq**(p,”Sam”) && **eq**(g,”male”) //fact 7

|| **eq**(p,”Mary”) && **eq**(g,”female”)//fact 8

|| **eq**(p,”Denise”)&& **eq**(g,”female”)//fact 9

;

}

// f is the father of c

**relation** father(**lref**<string> f,**lref**<string> c) {

//... if f is male and c is child of f

return gender(f,”male”)&& child(c,f);//rule2

}

Facts and rules are both declared as functions with the return type *relation*. The parameter types are specified in terms of template lref which stands for *logic reference*. Here it provides a facility similar to the pass by reference mechanism in C++. However, unlike references in C++, logic references can be left uninitialized. The value underlying a logic reference can be obtained by dereferencing it with operator \*.

Function eq is called the unification relation. Its job is to *attempt* to make its two arguments equal. If any one of its arguments is an uninitialized logic reference, it will assign the other argument to it. If both arguments have well defined values then it simply compares the two arguments. This task is referred to as ***unification***. Consider the call eq(c,"Sam") in relation child. If c has been previously initialized with a value, eq will compare the contents of c with "Sam". However if c has not been initialized, "Sam" will be assigned to c. The type relation, logic references and relation eq will be discussed further in sections 2.1, 2.2 and 2.3 respectively.

Neither eq nor any of the user-defined relations like child or gender return the results of their intended computation immediately when invoked. Instead they return function objects that encapsulate the intended computation. The function objects can be stored in an object of type relation. Their evaluation can be triggered by application of the function call operator on relations. Given the above C++ definitions for the relations, we are in a position to make some queries and assertions. The following is a simple assertion to check if Sam is male:

relation samIsMale = gender("Sam", "male");

if( samIsMale() )

cout << "Sam is male";

else

cout << "Sam is not male";

Similarly we can check if Frank is Sam’s father:

relation samsDadFrank = father("Frank","Sam");

if(samsDadFrank())

cout << "Frank is Sam's father";

else

cout << "Frank is not Sam's father";

We can also issue generative queries such as “What is Sam’s gender?”:

lref<string> g; // g not initialized

relation samsGender = gender(“Sam”, g);

if( samsGender() )

cout << "Sam’s gender is " << \*g;

else

cout << "Sam’s gender is not known";

Here we pass “Sam” as the first argument and simply leave the second argument undefined (i.e., not initialized to any value). Note how the same function gender is used to assert if Sam is male and also find out Sam’s gender. When all the arguments have been defined (i.e., initialized to some value) it performs a check to see if they satisfy the gender relation. Arguments that have been left undefined will act as output parameters and will be initialized with a value that satisfies the gender relation. This bidirectional nature of the lref parameters behaving as input or output is central to the logic programming model. It is important to note, however, that lref parameters on relations typically do not act as both input and output simultaneously as is often the case when using the pass-by-reference scheme in functions like swap.

What happens if both arguments to gender relation are undefined?

lref<string> p, g; // p and g not initialized

relation anyPersonsGender = gender(p, g);

if( anyPersonsGender() )

cout << \*p << "’s gender is " << \*g;

In this case both p and g will be assigned values by gender. Since the person-gender pair (“Frank”, “male”) is declared first in the gender relation, p will be assigned “Frank” and g will be assigned “male”.

Generative queries, such as samsGender and anyPersonsGender above, may have zero, one or many solutions. So far we have only generated the one solution from them. Iterating over all solutions, quite naturally, involves the use of a while loop instead of the if statement we have used so far. The previous example can be rewritten to print all persons and their genders as follows:

while( anyPersonsGender() )

cout << \*p << "’s gender is " << \*g << "\n";

Similarly for listing of all Frank’s children:

lref<string> c;

int count=0;

relation franksChildren = father("Frank", c);

while( franksChildren() ) {

++count;

cout << \*c << " is Frank's child\n";

}

// c is now back to its uninitialized state

cout << "Frank has " << count << " children";

Once all solutions have been exhausted, invoking franksChildren() returns false and causes the while loop to terminate. Also, when all solutions have been exhausted, logic reference c will be automatically reset to its original uninitialized state.

Notice how the ability to use fundamental imperative constructs like the if statement and the while loop makes the transition between the logic programming model (in which results are generated) and the imperative model (in which the results are consumed) simple and seamless.

Function eq, template type lref and type relation along with overloads for operators && and || provide the foundation for logic programming in Castor. They are described briefly in the following sections. For an in depth coverage of their design and implementation, refer to [[CastorDesign]](#_References).

## Type relation

In logic programming it is common to refer to facts and rules as **predicates** or **relations**. The term “relation” originates in set theory where it is used to imply an association between sets. So, gender is a ***binary relation*** between a set of individuals and the set of genders. Generally a strict distinction between rules and facts is not required when programming with Castor as they can be mixed freely within the same function/expression[[1]](#footnote-1). Keeping with logic programming lingo, we will henceforth refer to functions that represent facts or rules as **relations**. So, functions with return type relation are themselves referred to as relations.

The type relation internally represents a function or function object with no arguments and return type bool. Thus child, gender and father return a function object that can be evaluated later in a lazy manner.

## lref: Logic reference

Template type lref is an abbreviation for *logic reference* and provides a facility for passing values in/out of relations in the form of arguments, similar to references in C++. Unlike C++ references, a logic reference does not have to be initialized and provides the member function defined for checking if it has been initialized. The dereference operator \* and the arrow operator -> can be applied to access the value referred to by a logic reference.

Now let us understand the initialization semantics of lref. This is helpful reasoning about operational semantics of relational code. When an lref<T> is initialized (i.e., constructed) or assigned a value of type T (or a type convertible to T), it internally stores a copy of the value. When initialized with another lref<T> (i.e., copy constructed), both logic references will be *bound* *together*. References that are bound together will refer to the same underlying value (if any). Thus any change to the underlying value of one logic reference is observed by all logic references that are bound together. A binding between logic references cannot be broken. That is, if logic references *A* and *B* are bound together and *C* and *D* are bound together, then *C*’s binding with *D* cannot be broken in order to form a binding with *A* and *B*. C will continue to be a part of the binding for the duration of its lifetime. Logic references can only be bound by initialization (i.e., copy construction) and not by assignment. A binding can only be formed during construction of the logic reference and will be automatically broken when the logic reference is destroyed. When the last logic reference that is part of the binding is destroyed, it will deallocate the underlying value.

Starting with Castor 1.1, pointers to objects can also be used to initialize an lref. When using pointers we must specify whether the lref should manage the lifetime of the object referenced by the pointer. For example:

//lifetime of "Roshan" will be managed

lref<string> s(new string("Roshan"), true);

//lifetime of *name* will not be managed

string name="Naik";

lref<string> s2(&name, false);

Assignment with pointers is performed using method set\_ptr:

string str="Castor";

s.set\_ptr(&str, false); // deallocates "Roshan". Will not manage lifetime of str

Although using pointers to assign objects to lrefs is useful (especially when mixing paradigms) and efficient, it can also be dangerous if not used carefully. Great care should be taken when specifying the lifetime management policy. For instance, if the pointer refers to an object on the stack, the lref should not be requested to manage its lifetime. Similarly, if two independent lrefs are made to refer to the same object using pointer assignment / initialization, both should not be requested to manage the lifetime of the object. Programmer must ensure that the objects being handed over to lrefs using pointers continue to exist as long as the lrefs can attempt to access them. Also, accidentally specifying lifetime management policy to false when true is intended will cause memory leaks.

## Relation eq: The unification function

Function eq is the unification function and takes two arguments. The arguments may be logic references or regular values. eq returns an expression (i.e., function object) which when evaluated attempts to *unify* the two arguments. If unification succeeds it returns true; otherwise it return false. Unification performed by eq is defined as follows:

* If both arguments are initialized, their values are compared for equality and the result of comparison is returned.
* If only one argument is initialized, the uninitialized argument will be assigned the value of the initialized one in order to make them equal.
* If both arguments are uninitialized, an exception is thrown.

So unification will *generate* a value for the uninitialized argument to make both arguments equal, or it will *compare* its arguments if both are initialized. In short, unification is a “generate or compare” operation. It is possible to implement other variations to unification, but is generally not required.

The expressions returned by the various calls to eq within, for instance, the gender relation are stitched together to form a bigger compound expression using || and && operators. It is important to note that the resulting compound expression is returned without being evaluated. These expressions are evaluated in a lazy manner. In other words, these expressions are stored in an object of type relation and will be subject to evaluation in the future when needed. The following section examines the evaluation of these expressions in detail.

## Evaluating Queries

Given the above relations, we can formulate the query that tests for “Is Sam male?” and store it in a variable samIsMale as follows:

relation samIsMale = gender("Sam", "male");

The call gender(“Sam”,“male”)does not perform any real computation; it simply returns a function object that can be later executed to determine if Sam is male. Invocation of a relation does not execute/evaluate it. This splitting of invocation from execution is a case of *lazy evaluation*. In order to execute this expression we simply apply the function call operator on the variable that holds the expression:

if( samIsMale() )

cout << "Sam is male";

else

cout << "Sam is not male";

The gender relation was defined previously as follows:

// 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");

}

Since “Sam” and “male” were passed as arguments to gender, the expression returned by gender looks as follows with the arguments substituted:

**eq**("Sam", "Frank") && **eq**("male", "male")

|| **eq**("Sam", "Sam") && **eq**("male", "male")

|| **eq**("Sam", "Mary") && **eq**("male", "female")

|| **eq**("Sam", "Denise") && **eq**("male", "female")

This expression contains four && expressions joined by three || operators. If any one of these && expressions returns true, the overall expression has been solved and evaluation halts in order to report success. Let us step through the execution that takes place when operator() is applied to samIsMale. The first expression eq(“Sam”, “Frank”) && eq(“male”, “male”) is chosen for evaluation. An attempt to unify “Sam” with “Frank”, when evaluating eq(“Sam”, “Frank”), fails as “Sam” is not equal to “Frank”. Short-circuit evaluation principle tells us that the remainder of this && expression does not need to be evaluated. Thus, this path of execution is abandoned immediately and backtracking will resume execution from the next && expression eq(“Sam”, “Sam”) && eq(“male”, “male”). This time both halves of the && expression unify successfully as their arguments are equal. A solution to the expression has been found and true is returned to the caller which happens to be the if statement.

If the function call operator is applied once again to samIsMale, execution resumes from the point at which it was previously halted. In this case the third && expression is chosen for evaluation, which fails due to failure in unification of “Sam” and “Mary”. This leads to evaluation of the fourth and final && expression which also fails for similar reasons and false is returned to the caller. All && expressions have now been evaluated and applying the function call operator to samIsMale henceforth will immediately return false.

Now let us consider how to specify generative queries such as “What is Sam’s gender?”. Such a query is constructed by calling gender with first argument initialized to “Sam” and leaving the second argument uninitialized:

lref<string> g; // g not initialized

relation samsGender = gender(“Sam”, g);

if( samsGender() )

cout << “Sams gender is ” << \*g;

Notice how it is possible to use the same gender relation to both assert if someone is male and to find the gender of a given individual. When a solution is found, g will be initialized to “male”. The expression returned by gender(“Sam”, g)now looks like this once we substitute the arguments:

**eq**("Sam","Frank") && **eq**(g,"male")

|| **eq**("Sam","Sam") && **eq**(g,"male")

|| **eq**("Sam","Mary") && **eq**(g,"female")

|| **eq**("Sam","Denise") && **eq**(g,"female")

When operator() is applied to samsGender for the first time, the first && expression fails evaluation as “Sam” does not unify with “Frank”. Backtracking proceeds to try out the second expression eq(“Sam”, “Sam”) && eq(g, “male”). Unification of “Sam” with “Sam” succeeds and then eq(g, “male”) is evaluated. Since g is undefined, eq will assign “male” to g and thus unification succeeds. Evaluation of the entire expression halts and returns true to the calling if statement. If operator() is again applied to samsGender, execution will resume from the point it was previously halted. However, one very interesting thing happens before execution proceeds. It is critical that side effects occurring in the previous path of execution must be undone before attempting a different alternative. If side effects are not reverted before backtracking pursues another alternative, the side effects can affect the results of future evaluations and lead to incorrect results. Thus unification of g with “male” has to be reverted causing g to go back to its original uninitialized state. This undo feature is automatically provided by the unification function eq.

The above query has only one solution, but as we observed previously, queries can have multiple solutions. For instance, we may want to find all males in the system. Once again we resort to the gender relation, but this time we leave the first argument uninitialized and initialize the second argument to “male”:

lref<string> person;

relation males = gender(person, "male");

while( males() )

cout << \*person << " is male\n";

This time around we repeatedly invoke males() till it returns false. Each invocation of males triggers the search for the next solution, i.e., a suitable value for the logic reference person. Each time around the loop, person is assigned a different value representing a solution. When all solutions have been discovered, males() returns false, the loop terminates and backtracking will restore person to its original uninitialized state. Since person does not refer to anything after the while loop has terminated, any attempts to access the underlying value using operator \* or operator -> will throw an exception.

## Recursive Rules

Recursion is often essential when declaring rules in the logic paradigm. For the family relationships example, let us consider defining a rule for the “ancestor” relationship. Note that there can be arbitrary levels of parent-child relationships between an ancestor and a descendant. We can recursively define the rule as:

A is an ancestor of D if:

* D is a child of A, **OR**
* D is a child of some P **AND** A is an ancestor of P

This lends itself naturally into the following definition for the ancestor relation:

// Flawed recursive definition

relation **ancestor**(lref<string> A

, lref<string> D) {

lref<string> P;

return child(D, A)

|| child(D, P) && ***ancestor***(A, P);

}

However, the above C++ definition contains infinite recursion. The return statement contains a recursive call to the ancestor. In order to break the recursion we need to postpone the recursive invocation so that it only takes place if truly needed. Castor provides helper relation recurse for defining recursive rules. recurse takes the relation that needs to be called recursively and the arguments that need to be used for the call. It returns a function object which when evaluated leads to the actual recursive call to ancestor. The problematic call ancestor(A,P) is simply rewritten as recurse(ancestor, A, P):

relation **ancestor**(lref<string> A

, lref<string> D) {

lref<string> P;

return child(D,A)

|| child(D,P)

&& **recurse**(ancestor, A, P);

}

In the above example, recursion is performed on relation ancestor that is defined as a global function. Usage of recurse is the same for static member relations too. For recursing on non-static member relations, the this pointer needs to be provided in addition to the method name and arguments as follows:

**recurse**(this, &Type::method, ..args..)

## Dynamic Relations

Definitions of all relations described so far have been fixed at compile time. Relation gender, for instance, provides a definite list of name-gender pairs that does not change at run time. But when this gender information is available only dynamically (say from a file or database), we need an alternative mechanism to build the set of clauses for the relation. Here we have relation gender, from section 1, which statically defines all the information.

// statically defined relation

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");

}

Let us assume this information regarding each person’s gender has been read from a file or database into a list<pair<string,string>> called genderList. Let the first item in the pair be a name and the second be his gender. We can now define relation gender\_dyn, based on genderList as follows:

list<pair<string,string> > genderList = ...;

// dynamically building a relation

Disjunctions **gender\_dyn**(lref<string> p

, lref<string> g)

{

Disjunctions result;

list<pair<string,string> >::iterator i;

for( i=genderList.begin();

i!=genderList.end(); ++i)

result.push\_back(

**eq(p,i->first) && eq(g,i->second)** );

return result;

}

Here we use the type Disjunctions to dynamically build the set of OR clauses for the relation. Type Disjunctions is itself a relation that supports dynamic addition of clauses. Thus we can trigger evaluation on it using operator(). The return type of gender\_dyn has also been changed from relation to Disjunctions. This is optional but useful, as it implicitly conveys the dynamic nature of gender\_dyn to consumers. Conceptually Disjunctions is simply a collection of relations. When a Disjunctions instance is evaluated, the relations contained in it are treated as if there exists an operator || between each pair of adjacent relations. Relations can be added at the front or at the back of a Disjunctions relation using methods push\_back and push\_front. During evaluation, the contained relations are evaluated from front to back. Notice how an entire relational expression eq(p,i->first) && eq(g,i->second) is added to the Disjunctions.

Relations Conjunctions and ExDisjunctions are also provided for building relations dynamically. They correspond to operators && and ^ respectively. The relation gender is redefined below using Conjunctions and Disjunctions.

Disjunctions **gender\_dyn**(lref<string> p

, lref<string> g) **{**

Disjunctions result;

Conjunctions conj1 = eq(p,"Frank");

conj1.push\_back( eq(g,"male") );

Conjunctions conj2 = eq(p,"Sam");

conj2.push\_back( eq(g,"male") );

Conjunctions conj3 = eq(p,"Mary");

conj3.push\_back( eq(g,"female") );

Conjunctions conj4 = eq(p,"Denise");

conj4.push\_back( eq(g,"female") );

result.push\_back( conj1 );

result.push\_back( conj2 );

result.push\_back( conj3 );

result.push\_back( conj4 );

return result;

**}**

In summary, Conjunctions, Disjunctions and ExDisjunctions together provide a facility for defining relations dynamically. This ability also naturally makes them a facility for runtime metaprogramming in the Logic paradigm.

## Inline Logic Reference Expressions (ILE)

One may often encounter cases where the relations are simple operations to be performed on logic references using standard operators like +, -, etc. For instance, assuming the existence of a relation multiply for querying/asserting the product of two numbers, we can obtain the cube of a number as:

lref<int> n=2, sq, cu;

multiply(sq,n,n) && multiply(cu,sq,n)( );

cout << \*cu ;

Using multiply the square is generated and from the square we obtain the cube. How would the above code look for computing the expression n\*5-n/2+5? It is evident that writing out simple arithmetic expressions involving more than a couple operators gets verbose and unreadable quickly. Castor allows inline specification of expressions composed from standard operators and at least one logic reference. The cube example can now be rewritten more concisely:

lref<int> cu, n =2;

eq\_f(cu, n\*n\*n)();

cout << \*cu;

Relation eq\_f unifies its first argument with the result of *evaluating* its second argument, which is a function object. It is important to note that the value of n\*n\*n is not computed when it is passed to eq\_f. Since the expression is composed using an lref it is turned into a function object and then passed to eq\_f. The function object undergoes evaluation when the evaluation of eq\_f kicks in. At the time of evaluation, n must be initialized, as accessing the value of an uninitialized lref results in an exception. Arbitrary expressions, such as n\*5-n/2+5, composed of lrefs and common overloadable operators, can be used to construct function objects easily:

eq\_f(cu, n\*5-n/2+5)();

Such inline specification of expressions involving logic references is called ILE, short for “Inline Logic reference Expression”. ILEs come in handy in a variety of situations. Printing to console with relation write\_f is another example where they can be used. The following example demonstrates the use of ILE for printing two strings separated by a comma:

lref<string> ls=”Hello”; string s=”world”;

write\_f(ls + string(",") + s)();

Relation write\_f takes a function or function object as argument and prints the result of the evaluation of its argument to stdout. We use an ILE in the above example to conveniently instantiate a function object and pass it to write\_f. ILEs can be used as arguments to any relation provided by Castor suffixed with \_f.

If T is the result type of an ILE, then all overloadable operators defined over T other than comma, dereference operator \*, &, and -> can be used on lref<T> to produce an ILE. Currently ILEs lack support for mixing in objects of an arbitrary type T2 even if relevant operators are defined over T and T2. Thus,

write\_f(ls + ",")(); //compiler error

fails even though operator + is defined for arguments of type string and const char\*.

So far we have used ILEs to easily pass expressions to other relations. ILEs that produce a boolean value can be used to create simple relations inline. Consider the following relation that prints the result of the comparison of its arguments.

relation greaterLessEq(lref<int> n

, lref<int> cmpVal) {

return write(n) && write(" is ") &&

**(** **predicate(n<cmpVal)** && write("lesser")

|| **predicate(n>cmpVal)** && write("greater")

|| write("equal") **)**;

}

Here, the ILE expressions predicate(n<cmpVal) and predicate(n>cmpVal) are used to conveniently define comparison relations directly inline instead of defining global less or greater relations to do the same job. Similarly, predicate(n%2==0) can be used to create an inline relation to test for even numbers, or one could conceive more complex expressions like predicate(n\*2 >= cmpVal\*n/2). It is important to note that relations defined using ILEs do not have the ability generate solutions. They will only assert if the said condition is true/false. Thus, predicate(n<cmpVal) will not generate values for n or cmpVal. All logic references involved in a relation are required to be initialized at the time when evaluation of the ILE occurs; otherwise an exception will be thrown. The exception will be thrown directly by any uninitialized logic reference when the ILE attempts to dereference it for evaluation. A more detailed discussion on ILEs is presented in section 4.

## Sequences

Castor provides facilities for working with standard C++ sequence containers and iterators in a relational fashion. Tasks associated with sequences can be broadly categorized into those producing sequences and those iterating over the elements in it (i.e., consuming). Often, in traditional logic programming (as in functional programming or C++ template metaprogramming), operations requiring modifications to a sequence are performed by creating a new sequence reflecting the necessary changes. Deletion of elements in a sequence is done by producing a new sequence without the unwanted elements. Thus deletion of elements is a combination of iteration over the original sequence and producing another sequence. Addition of new elements can also be performed in a similar fashion. Modifying values of elements can be considered an operation on the element and not on the sequence itself.

Although it is possible to write relations where deletion or insertion of elements is reflected directly in the original sequence, we will restrict ourselves to traditional logic style techniques. The following sections describe some of the common tasks surrounding sequences.

### Generating Sequences

Sequences are created using the sequence relation. The following defines a relational expression that creates a list of three even numbers.

lref<list<int> > le;

relation evens = sequence(le)(2)(4)(6);

// see what it generates

if(evens())

copy(le->begin(), le->end()

, ostream\_iterator<int>(cout, " "));

The uninitialized logic reference le is passed as the first argument to relation sequence. The elements used for constructing the sequence are then consecutively passed individually to whatever is returned by the preceding function call. Since le is not initialized, when the sequence relation is evaluated in the condition of the if statement, le will be initialized to a list<int> containing values 2, 4 and 6. Note that the sequence relation automatically figures out (from the type of its first argument) if you are interested in creating a list<int> or vector<string> or some other sequence type. This ability is useful when writing generic code.

Abilities of the sequence relation go further. Sequences can be created not just out of simple values, but also from other sequences, logic references or an arbitrary mix of these. In the following code we create a list of strings using a mix:

string **s** = "One";

lref<string> **lrs** = "Two";

vector<string> **ls**;

ls.push\_back("Three"); ls.push\_back("Four");

vector<string> lsTemp;

lsTemp.push\_back("Five"); lsTemp.push\_back("Six");

lref<vector<string> > **lrls** = lsTemp;

// create the sequence into ln

lref<vector<string> > **ln**;

relation numbers =

sequence(**ln**)("Zero")(**s**)(**lrs**)(**ls**)(**lrls**);

// see what it generates

if(numbers())

copy(ln->begin(), ln->end()

, ostream\_iterator<string>(cout," "));

A current limitation when creating a sequence from other sequences is that, all sequences involved should be of the same kind. That is, lref<list<int> > cannot be created directly from a vector<int> or lref<vector<int> > and vice versa. However, this limitation is easily circumvented using sequence’s support iterators as follows:

vector<int> **vi**;

vector<int>::iterator **b1**, **e1**;

lref<list<int> > **lrli1**;

// generate using iterators to vi

relation r =

sequence(**lrli1**)(**vi.begin()**, **vi.end()**);

lref<vector<int> > **lrvi**;

lref<list<int> > **lrli2**;

lref<vector<int>::iterator> ***b2***, ***e2***;

// generate using lref<iterator> to lrvi

relation r = begin(***b2***, **lrvi**) && end(***e2***, **lrvi**)

&& sequence(**lrli2**)(***b2***,***e2***);

The relations begin and end will produce logical references that point respectively to the beginning and one past end of the sequence referenced by lrvi. The iterators are produced in a lazy fashion, i.e., only if and when they are actually evaluated in the future. The logic references produced by begin and end are subsequently provided to sequence for constructing lrli2. In general, care must be taken to avoid performing eager evaluations on logic references as they are typically initialized with appropriate values when backtracking and unification occurs.

### Iterating over sequences

Relations head, tail, next and prev are provided for iterating through sequences. Relations head and tail provide relational style iteration that is also similar to how iteration is performed in functional languages. Relations next and prev provide support for iterating with iterators that is closer to iteration techniques in traditional C++.

#### Iterating with head and tail

In this technique to iterate over all elements in a sequence container (like a list or vector), we split it into its head (i.e., the first element) and tail (i.e., collection of remaining elements). Splitting the tail again into its head and tail produces the second element in the original container. In this fashion we can continue to split the tail recursively till the tail is empty. The following example demonstrates the use of relations head and tail for printing all values in a list of integers.

relation printList(lref<list<int> > li) {

lref<int> h;

lref<list<int> > t;

return head(li, h) && write(h) && write(",")

&& tail(li, t)

&& recurse(printList,t);

}

list<int> li;

// .. add numbers to list ..

printList(li)();//print elements

Relation head\_tail is available for conveniently obtaining the head and tail directly in a single call. The return statement above can be rewritten using head\_tail as follows:

return head\_tail(li,h,t)

&& write(h) && write(",")

&& recurse(printList,t);

In cases where the tail is only used if a certain condition is true, then relations head and tail can be used more efficiently by computing the tail after evaluating the condition. Since computing a tail is an O(n) operation it makes sense to delay computing the tail after determining that it will be needed. A typical example of this is when searching for a vale in a list. Once the value of interest is found, the remainder of the list does not need to be processed. In cases like printList, where it is obvious that tail is always processed, head\_tail may be preferred for sake of brevity.

Similar to head and tail, there also exists head\_n and tail\_n when the first or last *n* elements in the sequence are of interest. The following example generates the first and last two items from a vector containing four items.

int a[] = { 1,2,3,4 };

vector<int> **v** (a+0, a+4);

lref<vector<int> > h;

lref<vector<int> > t;

head\_n(v, **2**, h)();

// now h contains {1,2}

tail\_n(v, **2**, t)();

// now t contains {3,4}

The extra argument is used to specify the number of head/tail items we are interested in. Exception will be thrown if the size of the sequence is less than the specified head/tail size.

#### Iterating with next and prev

Relation next represents a binary relation between a value and its successor. Since the successor of an iterator/pointer is another iterator/pointer that points to the next element in the sequence, we can use next to perform iteration. The following simple example prints the successor of 2:

lref<int> s;

next(2,s)();

cout << \*n;

next can also be used to generate a predecessor:

lref<int> p;

next(p,2)();

cout << \*p;

When both arguments are defined next will assert if the second argument is a successor of the first. An exception will be thrown if both arguments are undefined. Similar to next, relation prev is also available. The only difference between the two is that the order of arguments is reversed. next and prev can be used interchangeably depending on programmer’s preference of which one is more readable in a given context.

The following example demonstrates use of next to print all items bounded by a pair of iterators.

relation printAll( lref<int\*> beg\_

, lref<int\*> end\_ ) {

lref<int> val; // for storing \*\*beg\_

lref<int\*> n;

return predicate(beg\_==end\_)

^ ( dereference(beg\_, val)

&& write(val)

&& next(beg\_,n)

&& recurse(&printAll,n,end\_) );

}

int ai[]={1, 2, 3};

printAll(ai+0, ai+3)();

Relation printAll traverses through the items by recursively producing the successor to iterator beg. In each recursive step a check is performed to ensure that sequence is not empty by comparing beg to end. If there are elements in the sequence, we dereference beg (in a relational manner using dereference) to produce the underlying value and print it using write. After printing the first item, we proceed to recursing on the remainder of the sequence by producing successor of beg in n. The above example can also be rewritten in terms of iterators from the standard library, such as vector<int>::iterator, instead of simple pointer based iterators, by simply substituting occurrences of int\* with vector<int>::iterator in the above example.

#### Iterating with item

Relations next and prev are useful in cases when explicit control over the iteration process is required. In cases when the intent is to simply produce all the values in a sequence one by one, relation item provides a simpler alternative. item takes 3 arguments; the first two are a pair of iterators (or a pair logic references to iterators) representing the sequence and the third argument is logic reference. An element from the sequence is produced in the third argument for each evaluation of item. The first two arguments must be defined.

int ai[] = { 1, 2, 3, 4 };

vector<int> vi(ai+0, ai+4);

lref<int> val;

// 1 – iterating with regular iterators

relation r = item(vi.begin(), vi.end(), val);

while(r())

cout << \*val << ",";

// 2 - iterating with logic references to

// iterators

lref<vector<int>::iterator> lBeg = vi.begin()

, lEnd = vi.end();

r = item(lBeg, lEnd, val);

while(r())

cout << \*val << ",";

In the above code, the third argument is left undefined in order to produce values from the sequence. item can also be used to assert if a particular value is present in the sequence simply by defining the third argument to the value of interest:

if( item(vi.begin(),vi.end(),4)() )

cout << "found!";

### Unification of Collections

Unification facilities provided by relation eq as discussed in section 2.3 above is not limited to scalar value types. Unification can be performed on collection types in a similar fashion.

// produce a sequence

int a[] = { 1,2,3,4 };

vector<int> **v** (a+0, a+4);

lref<vector<int> > **lrv**;

eq(lrv,v)**()**;

assert(\*lrv==v);

// compare items

lref<list<int> > **lrl**= list<int>(a+0,a+4);

if(eq(lrl,v)())

cout << "\*lrl and v are equal” ;

Relation eq provides the basic unification support for collections (i.e., sequences). In section 2.8.1 we discussed how sequence can be used to generate sequences. When the first argument is not initialized, sequence generates a sequence filled with the elements specified in the remaining arguments. However, if the first argument is initilized, it will perform comparison of the elements in the first argument with the items from the remaining arguments. This makes sequence a powerful unification facility for sequences. We have already seen examples of producing sequences in 2.7.1. The following example demonstrates its use for comparison.

int a[] = {1,2,3};

lref<list<int> > **lrl**= list<int>(a+0,a+3);

lref<int> **lri**=1;

vector<int> **v**; v.push\_back(2);

relation r =

sequence(**lrl**)(**lri**)(**v**.begin(), **v**.end())

(3);

assert(r()); // evaluation succeeds

Unlike eq, sequence allows creation/comparison of sequences using any mix of other sequences, iterators, lrefs of other sequences etc.

### Summary

There are conceivably many ways of working with collections/sequences and Castor only provides support for a few useful ones. We can deal with collections directly as a whole or via iterators. Relations item, next, prev and deref are lightweight facilites which deal with collections using iterators. Relations eq, sequence, head, tail, item allow working with whole collections directly. sequence is the most feature-rich and flexible, but is also heayweight compared to the alternatives. Other relations such as empty, size, insert, merge, eq\_seq, etc., are also available for working with collections and iterators. Refer to the reference manual for a complete list.

## Cuts – Pruning alternatives

It is sometimes useful to discard paths that will not produce solutions or will produce duplicate solutions when backtracking occurs. Such explicit pruning of paths is often done for efficiency reasons. There is no reason to waste time on pursuing alternative paths that are known to fail or not produce anything of interest (e.g., duplicates of previously found solutions). Consider the following relation that searches through a binary search tree for a given value:

// Binary search tree

struct BST {

BST\* l; // left subtree

BST\* r; // right subtree

int value; // current node

...

};

relation b\_search(lref<int> val

, const BST\* tree) {

return

predicate(val**==**tree->value)

|| predicate(val**<**tree->value)

&& recurse(b\_search, val, tree->l)

|| predicate(val**>**tree->val)

&& recurse(b\_search, val, tree->r)

;

}

Relation b\_search consists of three clauses. The first clause simply checks if the value of the current node matches the argument. The remaining two clauses will search recursively on the left and right sub trees depending on how val compares with the current node. If the first equality comparison succeeds, it is evident that the remaining clauses can be discarded. Once any of the comparison operations succeed we would like the backtracking mechanism to stay committed to the current clause and ignore other alternatives. The facility for eliminating alternative paths is typically called a *cut* in logic programming. Class cut and relation cutexpr provide support for cuts in Castor. We can rewrite the above relation using cuts as follows:

relation b\_search(lref<int> val

, const BST\* tree) {

return **cutexpr(**

predicate(val==tree->val) && **cut()**

|| predicate(val<tree->val) && **cut()**

&& recurse(b\_search, val, tree->l)

|| predicate(val>tree->val)

&& recurse(b\_search, val, tree->r)

**)**;

}

Each occurrence of cut() marks the point at which we decide to commit to the current clause. The location of a cut() is called a *cut point*. A cutexpr only marks the boundaries within which pruning of alternatives takes place. In this case cutexpr(…) spans the all three clauses in the relation. Once the execution reaches one of the cut points, it will stay committed to the path beginning from the start of the cutexpr to the cut point. Thus all other alternatives available after “cutexpr(” begins will be discarded (or cut out) from consideration by the backtracking mechanism. Cuts do not affect the alternatives that existed prior to the cutexpr or the alternatives that exist after a cut point. The cutexpr itself merely provides the scope or the extent within which the cut operation takes place.

A cut point without a surrounding cutexpr, or a cutexpr without any cut points are both meaningless. Such occurrences can sometimes occur accidentally when removing cuts from a relation that currently makes use of cuts. By design, such mismatched occurrences will produce compilation errors. Following usage of cuts wherein a cutexpr(…) appears in the caller and a cut() appears in the callee is also not allowed:

// Error: cannot dynamically nest cuts

relation outer(...) {

return **cutexpr(** inner(..) || ... **)**;

}

relation inner() {

return ... && **cut()** ...

}

Excessive usage of cuts in logic programming is generally discouraged since they tend to make logic programs less readable. Also, when not used with care, they can incorrectly prune out valid paths. Their usage should be preferred primarily in situations where it leads to a reasonable gain in performance. It is desirable for a relation that uses cuts, to also produce the same results when the cuts are removed. Cuts that do not alter the results of a relation are referred to as *green cuts*. Cuts that are not green are called *red cuts*. Many usages of cuts can be rephrased more elegantly using the relational ex-or operator.

## Relational Ex-Or operator

We have seen operators && and || used to define relations. Castor also provides support for defining ex-or semantics between clauses in a relation using the operator **^**. It is useful in expressing the idea that the second clause should be attempted only if the first clause fails. Let us revisit the greaterLessEq example.

relation greaterLessEq(lref<int> n

, lref<int> cmpVal) {

return **predicate(**n<cmpVal**)** && write("n<cmpVal")

|| **predicate(**n>cmpVal**)** && write("n>cmpVal")

|| write("n==cmpVal");

}

The following usage of greaterLessEq exposes a problem in the above implementation.

relation r = greaterLessEq(2,3);

while(r());

This produces the following output:

n<cmpVal

n==cmpVal

This is because the while loop forces the backtracking to purse all remaining paths even after the first clauses matches. The second clause fails due to the **(**n<cmpVal**)** guard at its beginning and this leads to the evaluation of the third and final clause. Since the third clause does not have the guard (n==cmpVal), it succeeds and thus we observe the output “n==cmpVal”. An obvious solution is to put the guard in front of the third clause. Another solution is to introduce cut points at the end of the first and second clauses. However a simpler and preferred solution is to rewrite it in terms of the **^** operator as follows:

relation greaterLessEq(lref<int> n

, lref<int> cmpVal) {

return **(** predicate**(**n<cmpVal**)**

&& write("n<cmpVal") **)**

**^** **(** predicate**(**n>cmpVal**)**

&& write("n>cmpVal") **)**

**^** **(** write("n==cmpVal") **)**;

}

Here we are being explicit about the fact that three clauses are mutually exclusive candidates. Not only is this more explicit but also more readable and efficient. Once any one of the clauses succeeds greaterLessEq also succeeds. Backtracking will ignore any remaining unevaluated clauses. Thus all future attempts to seek more solutions from greaterLessEq will fail immediately. It is important to note the use of additional brackets around the clauses separated by ^. Since, unfortunately, the precedence of the ^ operator is higher than && and || operators, use of these brackets is necessary to preserve correct associativity.

Most, but not all, common usages of cuts can be elegantly replaced with use of ^ operator. Support for operator ^ to define relations is a feature unique to Castor.

## Specifying Lref parameters types for relations

There are basically three ways of specifying lref parameters for relations:

* Basic: lref<T>
* By reference: lref<T>&
* By const reference: const lref<T>&

**Basic:** If the parameter type is specified to be lref<T>, arguments of type T and lref<T> are both acceptable. This versatility makes this mechanism a common choice for specifying parameters on relations. When the caller passes an argument of type lref<T>, the callee receives it such that the new lref points to the same underlying object of type T. This is similar to passing pointers in C++. However, when the argument is an object t of type T, the callee receives a copy (on the heap) of t to which the callee’s lref refers to.

relation foo(lref<int> i) {

return True();

}

lref<int> li=2;

foo(1); // OK! passes a copy of 1

foo(li); // OK!

This basic parameter specification style is used most commonly in Castor.

**By reference:** For certain types where copy construction can be potentially very expensive such as std::vector, it is desirable to prevent implicit and (possibly) repeated copy construction. This can be done by disabling the ability to pass arguments of type T directly and requiring only arguments of type lref<T>. This can be done by specifying the parameter type as lref<T>&.

relation foo(lref<vector<int> >& v) {

return ...;

}

vector<int> v= ...;

lref<vector<int> > lv= ...;

foo(v); // Error!

foo(lv); // OK!

All relations in Castor that expect container types (such as empty and size) as arguments use this form of parameter specification.

**By const reference:** This mode of parameter specification is similar to the basic style with one significant difference. A const lref<T>& parameter cannot be passed as argument to another relation that specifies its lref parameter by reference.

E.g.:

relation bar(lref<int>& j) {

return ...

}

relation foo(const lref<int>& i) {

return bar(i); // Error!

}

Yet another, but minor, difference compared to the basic style is that the value of the lref cannot be modified *directly* inside the relations:

relation foo(const lref<int>& i) {

i=2; // Error!

return bar(i);

}

This is rarely an issue since direct mixing of such imperative code inside a declaratively specified relation should be avoided in practice. The reason being, the imperative assignment of 2 to i here is performed when the relation is invoked as opposed to when the relation is evaluated.

## Debugging

Unfortunately, in an imperative world, debugging relational code is not as straight forward as debugging imperative code. Debuggers for C++ are, for good reason, designed to debug imperative code conveniently. In principle, since we let the computer figure out how to solve the problem, we should be oblivious to how evaluation takes place. In practice, however, it is useful to be able to peek into the evaluation as it progresses. This is important for verifying correctness of user-defined relations especially when the expected output does not match what is observed.

For debugging logic programs some creative thinking is typically required. Since relations return expressions that will be lazy evaluated, it is not very interesting to place a breakpoint directly inside the body of a relation’s definition. In fact, some of the relation’s lref arguments may not even be initialized with a value at the point where the relation is called, but are likely to be initialized by the time the relation undergoes evaluation. Also most of the actual execution occurs within the function objects returned by operators || and && and relation eq and debugging using breakpoints inside these library artifacts requires understanding of Castor’s implementation details[[2]](#footnote-2).

A simple and primitive technique is to insert print statements at relevant points to observe the progress. Castor provides a relation write for printing to stdout in relational manner. In the following example we add debug statements to the ancestor\_dbg relation to observe the generation of values P in the second clause as an attempt to find a solution progresses.

relation ancestor\_dbg(lref<string> A

, lref<string> D) {

lref<string> P;

return child(D,A)

|| child(D,P) && write(P) && write(",")

&& recurse(ancestor, A, P);

}

The following attempt to find all ancestors of Sam:

lref<string> X;

relation a = ancestor\_dbg(X,"Sam");

while(a())

cout << " :" << \*X << " is Sam's ancestor\n";

produces the following output on screen:

:Mary is Sam's ancestor

:Frank is Sam's ancestor

Mary,Frank, :Gary is Sam's ancestor

Gary,

It can be observed from this output that the ancestors Frank and Mary are discovered without resort to recursion since they are direct parents of Sam. Once both parents of Sam are found, the printed output reveals that backtracking proceeds to evaluate the recursive clause looking for Mary’s ancestors.

# Implementing relations imperatively

The ability to easily reach out from any paradigm to another, as and when required, is essential in multiparadigm programming. So far, we have seen examples of how relations can be built on top of other relations and also how relations can be consumed by imperative code. In this section we complete this cycle by understanding how relations can consume non-relational facilities. This involves using imperative techniques to implement the relation.

Relations may be defined either declaratively or imperatively. The examples seen so far in this document are cases of defining them declaratively since the programmer is not involved in providing the operational semantics for the relation. That is, the author of the relation does not specify the algorithms and data structures used to perform computation.

Producing imperative definitions for relations is generally more work compared to producing declarative definitions. Also declarative definitions are easier to read and get right. However, there can be situations that motivate a programmer to define relations imperatively. Typical reasons include:

* Interaction with low level (e.g., I/O, memory, drivers, etc.) or other imperative facilities for which relational abstractions either do not exist or the ones available are inadequate.
* Finer grain control over execution to improve performance when necessary.
* Improving the ability to step through a relation’s execution with a debugger.

Below we cover three approaches to implementing relations imperatively. The choice of which approach can be taken depends upon the complexity of relation.

## With relation predicate

A test-only relation is one that does not modify any of its arguments and consequently does not induce any side effects into the system that need to be reverted during backtracking. Such a relation is useful in testing if a given condition is true. Generally, due to their very nature, test-only relations succeed only once at most. Relation predicate is useful for defining such relations. The actual test condition can be packaged into a regular function that returns bool, and then invoked via relation predicate. Consider defining a relation that checks if a given file exists.

bool fileExists\_pred(string fileName) {

if(/\* file found on disk \*/)

return true;

return false;

}

relation file\_exists (lref<string> fileName\_) {

return predicate(fileExists\_pred,fileName\_);

}

This is essentially a two-step process. First, the predicate function fileExists\_pred contains the imperative code to be used for the relation. Second, relation predicate is used to convert the predicate function into a relation inside the “wrapper” relation file\_exists. Note that the relation takes an lref argument whereas the function takes a string argument. Although for both the parameter types can be lref<string>, it is natural for a relation’s parameter type to be an lref and that of an ordinary function to be a regular type. Relation predicate assumes that the target predicate function does not take an lref argument. Consequently, if filename\_ is a logic reference, it automatically dereferences fileName\_, when invoking the predicate function fileExists\_pred. If filename\_ is not a logic reference, it will be passed directly.

Also note the customary use of ‘\_’ at end of the parameter’s name ‘fileName\_’. This indicates that it is not a bi-directional argument. Thus its value should be defined by the time the evaluation of the relation takes place. If it is not defined at this time, an InvalidDeref exception is thrown when predicate attempts to dereference fileName\_.

## With relation eval

.. *todo*.. see reference manual.

## As coroutines

Every relation is actually an instance of some function object type. So far we have defined relations as functions and expressions. Since this syntactic approach is a declarative way of specifying the operational details of the underlying function object, its type name and imperative definition are not visible to us. In this section we shall see how to implement these function objects directly. This has the advantage that it gives complete control over the computational steps involved and also the relation is easier to step through using a debugger. The downside, however, is that it requires more code to implement and also needs some extra care.

We have seen that lazy evaluation is a key aspect of how relations operate. Each time a result is generated, its execution is suspended and control returns to the caller. On the next invocation, the relation resumes from its execution was suspended, produces the next value and once again suspends execution and returns control back to the caller. The caller can choose to invoke it again if more results are needed. Functions that support this *suspend-and-resume* behavior are called coroutines. This is unlike typical functions (i.e. subroutines) that always start execution from the beginning on every invocation and return control to the caller after generating all results. Since in case of coroutines, the caller and the callee both resume execution when control is transferred to them, coroutines can be considered to maintain a sibling relationship with their caller. Subroutines, on the other hand, can be considered to maintain a child relationship with their caller, as they undergo one full lifetime on each invocation by the caller.

Since C++ does not natively support defining coroutines, Castor provides the class Coroutine and four macros (co\_begin, co\_end, co\_yield, co\_return) for this purpose. Although these are designed for implementing relations as classes, one may use them outside the context of logic programming.

In Castor, a coroutine is implemented as a function object. To define this class, we derive it from Coroutine and implement the function call operator bool operator()(void) as follows:

// relation to check or generate values in a specified inclusive range

template<typename T>

class Range\_r : public Coroutine {

lref<T> val, min\_, max\_;

public:

Range\_r(lref<T> val, lref<T> min\_, lref<T> max\_)

: min\_(min\_), max\_(max\_), val(val)

{ }

bool operator() () {

co\_begin();

...

co\_end();

}

};

Note the use of macro co\_begin to start and macro co\_end to end the body of operator(). No statement should precede or follow these two macros in the method body. These two macros merely set up a switch statement spanning the definition of operator(). The complete definition of operator() is as follows:

bool operator() () {

co\_begin();

if(val.defined())

**co\_return(** (\*min\_<\*val && \*val<\*max\_)

|| (\*min\_==\*val) || (\*max\_==\*val) **)**;

for(val=min\_; (\*val<\*max\_)||(\*val==\*max\_)

; ++val.get())

**co\_yield(**true**)**;

val.reset(); // Important for backtracking

co\_end();

}

Notice the absence of return statements. These have been replaced with co\_return and co\_yield. Both macros return the evaluated value of their argument back to the caller.

Macro co\_yield indicates a point where execution is temporarily suspended, and a true/false value produced by evaluation of its argument is returned back to the caller. Next invocation of operator() will *resume* execution inside the method starting directly at this *yield* point. Invoking this macro essentially causes the coroutine to remember the point where execution had reached in the previous invocation. On the other hand, invoking co\_return indicates that no future resumption of execution is required from inside the method body, and the true/false value produced by evaluation of its argument is returned to the caller. All future invocations of operator() on this instance of the class will return false immediately.

Conceptually, co\_yield indicates temporary *suspension* of execution and co\_return indicates *completion* of execution. Since, in logic programming, returning false to the caller is an indication by a relation that its task is complete, co\_yield(false) also indicates completion as it returns false. When co\_yield’s argument evaluates to false, all future invocations of operator() will return false immediately. This behavior is similar to co\_return(false).

The definition of operator() first checks to see if the val is currently initialized in order to determine whether the relation needs generate a value for val or simply check if val lies in the specified range. If val is initialized, then only the check needs to be performed; otherwise, values need to be generated for val. The body of the if statement compares val with min and max and returns the result back to the caller. Since in this *test* mode there is no more work to be performed, co\_return is used to return the result of the comparison.

In the case that val is not initialized, the for loop is executed. The loop header generates values for val starting from min up to max. On every iteration, the body of the loop *yields* true back to the caller indicating successful evaluation of the relation. Once the execution enters the loop, on every subsequent invocation of operator(), execution directly resumes at co\_yield and thus performs one more iteration of the loop. Each time val is incremented and true is returned back to the caller. Once val exceeds max, the loop terminates and val is reset back to its original initialized state. Resetting val back to its original state once the relation’s work is done is important since backtracking requires all relations to revert their own side effects prior to signaling completion. Delegating the task of reverting side effects to the individual relations that induce them allows the backtracking subsystem to simple and efficient.

When the function object is a generic class that requires type parameters it is customary to provide a helper generic function to allow automatic template parameter deduction as follows:

template<typename T>

Range\_r **range**(lref<T> val, lref<T> min\_

, lref<T> max\_) {

return Range\_r<T>(val,min,max);

}

One important thing to keep in mind when implementing coroutines is to avoid defining variables inside operator(), since their state will not persist across invocations. Thus local variables required by the coroutine should be promoted to data members of the function object.

# Inline Logic Reference Expressions

An ILE is an expression composed of at least one logic reference variable and most of the common overloadable operators. Unlike typical expressions in C++, ILEs do not undergo evaluation to produce a result immediately at the point of definition. Instead, they produce an expression tree that represents the semantics of the expression. This expression tree can be evaluated to produce a result, at a later point in time, by applying the function call operator. The following example demonstrates these basic semantics:

int i=0; // plain old variable

i+1+1; // simple expression: produces value 2

lref<int> li=2; // logic reference

(li+i)\*2; // ILE: produces an expression tree

Following example uses an ILE where a function is required.

template<typename Func>

void printResult( Func f ) {

cout << f();

}

printResult( li\*li / 2 ); // pass ILE as argument

In the above code, the ILE undergoes evaluation inside the printResult when f() is evaluated. We can also return functions from other functions:

void runtests();

boost::function<int()> halfOf( lref<int> li ) {

return li/2; // return an ILE

}

boost::function<int()> f = halfOf(4);

cout << f(); // ILE is evaluated here

ILEs provide a convenient way to create expressions that require delayed evaluation. The convenience comes from not having to package them explicitly inside a named function. Such expressions are also traditionally called *lambda expressions*. Every ILE contains a copy of all variables and values required to compute its expression. This set of variables and values is referred to as the ILE’s *closure*. Since copying logic references is like copying pointers or references (i.e., the copy refers to the same underlying object), any changes to the object referred to by the original lref will be observable to the ILE. When multiple lrefs refer to the same object, the object will be kept alive until the reference count goes down to zero. This makes it safe to evaluate ILEs even after the termination of the scope in which the ILE was created. This safety is essential in getting the most out of the delayed evaluation semantics provided by ILEs. The following examples illustrate the closure semantics.

int x=2;

lref<int> lx=3;

boost::function<int()> ile = lx+x;

cout << ile(); // prints 5

x=1; // this will not observable to the ile as it contains a copy of x

cout << ile(); // prints 5

lx=4 ; // updating the lref will be observable to the ile

cout << ile(); // prints 8

Since an ILE is an expression and not a function, it does not accept any arguments at the time of evaluation. It is a free -standing expression that can be evaluated at anytime and as many times as needed. As we saw above, each evaluation may potentially produce a different result. Changes in the result could occur due to external code altering the object referred to by one of the logic references, as in the code above. A change in the result could also be due to the ILE inducing side effects on its own closure as follows:

lref<int> lx=3;

boost::function<int()> ile = ++lx;

cout << ile(); // prints 4

cout << ile(); // prints 5

cout << ile(); // prints 6

ILEs such as ++lx which induce side effects are said to be *impure*. Those that don’t, such as x+2, are *pure*. [[3]](#footnote-3)

## Creating relations from ILEs

ILEs were originally devised as a quick and easy way to create anonymous relations directly inline from simple expressions instead of having to create named relations. ILEs that return bool values can be turned into relations with the help of relation predicate. Relation predicate is an adapter that allows functions or function objects (with up to six arguments) that return bool to be treated as relations. Relations created by passing an ILE to predicate are referred to as ILE-based relations.

The following example demonstrates the use of ILE to generate even numbers.

//Print all even numbers in the inclusive range 1...20

lref<int> x;

relation evens = range<int>(x,1,20)

&& predicate(x%2==0);

while(evens())

cout << \*x << " ";

Above, the ILE-based relation predicate(x%2==0) tests if x is even. We start out with x not being initialized. Relation range generates values for x, and since x is part of the ILE’s closure, any changes to x will be visible to the ILE. This allows the ILE-based relation to test the values one by one as they are generated. This is an example of the classic generate-and-test pattern commonly used in logic programming. In both the examples relation range is used to generate values for logic references and the ILE-based relations filter out values that fail the check.

The following example demonstrates the use of ILE to generate Pythagoras triplets.

// Print all Pythagoras triplets less than 30

lref<int> x,y,z; int max=30;

relation pythTriplets =

range<int>(x,1,max)

&& range<int>(y,1,max)

&& range<int>(z,1,max)

&& **(**  predicate(x\*x+y\*y==z\*z)

|| predicate(z\*z+y\*y==x\*x) **)**;

while(pythTriplets())

cout << \*x << "," << \*y

<< ","<< \*z << "\n";

Here the first three uses of range generate values for x, y and z in the range 1 through 30 and the two ILE based relations are used to check if x, y and z form a Pythagoras triplet.

Another common usage of ILEs is with relation eq\_f, which takes a function or function object as its second argument. It unifies the first argument with the value obtained by evaluating the second argument. Thus, if x and y are lrefs, then eq\_f(x, y\*3) unifies x with the value obtained by evaluating the ILE y\*3.

Impure ILEs should not be used when creating ILE-based relations. If such relations produce side effects, they will not be reverted during backtracking and will most likely lead to surprising results. Consider predicate(++x<5) which increments value of x each time it is evaluated. Here, ++x<5 is an impure ILE as it modifies value of x. Backtracking relies on reversal of side effects. When such side effects are desired they should be packaged into a named relation that ensures reversal of the side effects. Relation predicate(++x<5) can be rewritten safely as inc(x) && predicate(x<5) or next(x,y) && predicate(y<5). The former increments x, and the latter unifies y with the incremented value of x (without effecting value of x). The latter style is generally preferred.

The advantage of ILE-based relations is the brevity; the downside, however, is that they are not as full featured as named relations. ILE-based relations can only perform tests on values produced by other relations; they are unable to generate solutions themselves. Thus all logic references involved in the ILE must be initialized at the time of evaluation.

## Limitations of ILEs

There are a couple of limitations to ILEs that users should be aware of. The first one is due to language limitations, and the second one is by design.

**1) Inferred return type**: A seemingly trivial question that one may ask is “What is the type of the result produced on evaluation of an ILE?” The answer of course is “It depends on the expression represented by the ILE”. A more precise answer should be “Same as if the expression were rewritten with logic references substituted with their effective types”. That is, the expression x\*y should yield an object of the same type and value regardless of whether x and y are lrefs or plain old variables. Unfortunately, this is not always true as it is not possible to programmatically compute the return type of an arbitrary expression or function call in C++. Then how does one determine the return type of an ILE? It is inferred using the following intuitive rules:

* All comparison operators (<, >=, ==, != , etc.) and operators &&, || and ! have return type bool.
* Return type of prefix ++ and –– is T&, if T is the argument type.
* All other unary and binary operators are assumed to have return type same as the type of their first argument.

An easy way of determining the type is to simply observe whether or not the ILE as a whole represents a comparison operation. If the ILE finally yields the result of a comparison, then its return type is bool, otherwise the return type is determined by the first argument to operator that is evaluated last in the ILE. Here are some examples:

lref<int> x=3;

double y=1;

2.0 \* x; // double

x \* 2.0; // int

y\*x; // double

x\*y; // int

2\*x + 3.1; // int

3.1 + 2\*x; // double

5 < x; // bool

x == x; // bool

x < 3\*x+5; // bool

3\*x+5 < x; // bool

++x; // int&

**2) Supported operators:** All overloadable operators with the exception of the following are supported for the creation of ILEs.

* AddressOf operator &
* Dereference operator \*
* Member access operator ->
* Indexing operator []
* Comma operator ,
* All forms of assignment (=, +=, \*= etc.)

Since assignment is used to assign a new value to an lref immediately, it obviously cannot be used to produce ILEs. Similarly, dereference operator \* and operator -> are used to access the underlying value and it rules them out from being used to compose ILEs. Address-of and comma operators take on default language -defined semantics when used on lrefs. Finally, as there is no reasonably intuitive way to know the result type of an index operator, it too remains unsupported.

## Summary

The following is a gist of the major points covered in this section on ILEs:

* ILEs are expressions that can be treated as function objects.
* Every ILE carries its closure with it. Thus, it can be safely evaluated even after the termination of their lexical scope.
* ILEs are either pure (i.e. free of side effects) or impure.
* Only pure ILEs should be used to create ILE-based relations.
* ILE-based relations cannot generate solutions; they can only perform testing of values generated by other relations.
* Return types of ILEs are determined by applying the two rules noted above.
* Most overloadable operators are supported on lrefs for creating ILEs.

# Limitations of Logic Paradigm

## Bi-directionality of lref arguments.

Typically lref parameters for relations are bi-directional. A relation generates a value for an lref parameter if its value is left undefined by the consumer of the relation. However it is not always feasible or even sensible for a relation to generate values for some of its lref parameters. Values for these lrefs must be defined by the time the relation is evaluated. Such lref parameters are called *input-only* parameters.

Consider implementing relation not\_eq which succeeds if the two arguments are not equal.

relation **not\_eq**( lref<int> lhs\_

, lref<int> rhs\_) {

return …

}

In this case, it is straightforward to compare the two arguments and check if they are not equal. But if one of the arguments is not initialized, on what basis could we generate a sensible value(s) for such that it is not equal to the other? There are multiple possibilities but no single approach can be easily deemed to be any better than another. For instance, we could start at 0 and go on till we hit numeric\_limits<int>::max(). Or we could start at numeric\_limits<int>::min() continuing until numeric\_limits<int>::max(). Perhaps even backwards starting from numeric\_limits<int>::max().

Attempting a search in such a wide-spanning range is rarely a very productive or useful way to solve problems. The futility of such an approach will seem even more obvious if we consider defining not\_eq for string. Thus it only makes sense for lhs\_ and rhs\_ to be input-only parameters. By convention, names of input-only parameters contain a ‘\_’ suffix, and in-out parameters do not.

Another such example is the relation size provided by Castor.

relation **size**(lref<Cont>& cont\_

, lref<typename Cont::size\_type> sz) {

return ...

}

The first argument is any standard container and the second argument represents the size of a container. It is straightforward to test if cont\_ has size sz and also easy to generate a value for sz given an arbitrary container object. But, what if sz is initialized and cont\_ is not? It makes no sense to come up with random containers having size sz with arbitrary elements. Thus we specify cont\_ as an input-only parameter.

In each of these cases, it is left up to the consumer of these relations to decide how the values for an input-only parameter is to be generated.

## I/O is not reversible.

Backtracking by its very nature depends upon the ability to revert side effects when pursuing alternate paths of evaluation. I/O, on the other hand, is rarely something that can be reverted. Once some data has been written to a socket or a document has been printed, there is no going back. Thus I/O and backtracking are at odds with each other. There are two ways to make these co-exist. First option is demand support from all kinds of I/O to be reversible. This is clearly not practical. The second option is to turn a blind eye to any I/O that occurs during the evaluation of relations. This is the only practical way in which the two can be made to co-exist. However, it is up to the consumer of any relation that performs I/O to ensure that such side effects do not impact or interfere with the choices made during backtracking. They should be used in a way that is “backtracking-safe”.

Relations write and read are examples of I/O relations in Castor. Consider the following misuse of read to check if the word read from stdin is either “Hello” or “World”.

relation r = read("Hello") ^ read("World");

Although the declarative reading “read either Hello or World” is sound, it assumes that if the first read fails then its side effects will be reverted so that the second read can re-read the data. The correct way to perform this operation is to call read only once as follows:

lref<string> w;

relation r = read(w) &&

(eq(w,"Hello") ^ eq(w,"World"));

Here we read the value, whatever it may be, into w, and then check if w is either “Hello” or “World”.

# Logic Programming Examples

In this section we solve a few problems using LP to get a better feel for logic programming techniques.

## Factorial

lref<int> j;

relation r = range(j,1,10)

>>= reduce(j, std::multiplies<int>());

r();

cout << "Factorial of 10 = " << \*j;

## Directed acyclic graphs.

1

3

4

5

2

Figure 1

Here we consider the problem of finding paths in the above directed acyclic graph consisting of 5 edges. The graph can be easily described in terms of its edges. In the following edge relation, each line represents a directed edge of the graph. The parameters n1 and n2 represent the start point and the end point of an edge respectively.

// Representation of the directed edges in the DAG shown in figure 1

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)

;

}

Given the definition of an edge we say that a path exists between two nodes Start and End if:

* A directed edge exists between Start and End. **OR**
* A directed edge exists between Start and some node (nodeX) **AND** a path exists from nodeX to End.

This relation can thus be defined as follows:

// Rule defining the meaning of a path

relation **path**(lref<int> start, lref<int> end) {

lref<int> nodeX;

return edge(start, end)

**||** edge(start, nodeX)

&& recurse(path, nodeX, end);

}

Given the relations edge and path above, we are in a position to ask many different questions concerning paths in the directed acyclic graph. A simple question is to see if a path exists between any two nodes:

relation p1\_4 = path(1, 4);

if(p1\_4())

cout << "Path exists from 1 to 4";

A typical problem in graphs is to find all nodes reachable from a particular node. The following code prints all nodes reachable from node 5.

lref<int> node;

relation p5 = path(5, node);

while(p5())

cout<< \*node <<" is reachable from 5\n";

By leaving variable node uninitialized above, we are letting the computer generate values for it. It is interesting to note that 4 is listed twice when the above code is executed. That is due to the fact that node 4 can be reached via two independent paths from node 5, and during the process of inferencing both paths are discovered. Similarly, leaving the first argument uninitialized and specifying 5 as the second argument, gives a list of all nodes from which node 5 can be reached.

It is also possible to print all pairs of nodes between which a path exists by simply leaving both arguments to path uninitialized:

lref<int> n1, n2;

relation p = path(n1, n2);

cout<< "Path found between these nodes\n";

while(p())

cout << "(" << \*n1 <<","<< \*n2 << ") ";

The above code will print the following pairs in order: (1,2) (2,3) (3,4) (5,4) (5,2) (1,3) (1,4) (2,4) (5,3) (5,4). Once again the pair (5,4) is listed twice as two paths were found from 5 to 4.

## Finite Automata (FA)

Finite automata can be represented in the form of a set of transitions and a set of final states. Class Nfa below encodes the nondeterministic finite automata (NFA) from figure 2 which is equivalent to the regular expression ((ab)\*ba) | b. It has two static member relations transition and final. Relation transition describes each transition in terms of the two states and the input character that make up the transition. Relation final specifies the final states in the NFA.

A global relation run defines the rule for a successful run of any FA over an input string. The specific Finite Automata over which it operates is specified as a template parameter. The run defines that a run is successful if:

* The input string is empty **AND** a final state has been reached, **OR**
* There exists a transition from current state (st1) to some other state (st2) over the first character in the input string **AND** there is a successful run over the rest of the string starting from state st2.

Figure 2

b

a

b

b

a

**RegEx: (**(ab)\*ba**)**|b

Start State: 0

Final States: 3, 4

Class NFA’s static member relation transition describes the state transitions and member relation final describes the final states. Global relation run takes NFA as a template parameter. It executes the NFA over an input string by matching one of the possible transitions to the next character.

class **Nfa** **{** //=> ((ab)\*ba)|b

static const char a = 'a', b = 'b';

public:

// transitions NFA

static relation **transition**( lref<int> st1

, lref<char> ch

, lref<int> st2) {

return

eq(st1,0) && eq(ch,a) && eq(st2,1)

|| eq(st1,0) && eq(ch,b) && eq(st2,2)

|| eq(st1,0) && eq(ch,b) && eq(st2,4)

|| eq(st1,1) && eq(ch,b) && eq(st2,0)

|| eq(st1,2) && eq(ch,a) && eq(st2,3)

;

}

// all final states of the NFA

static relation **final**(lref<int> state) {

return eq(state,3) || eq(state,4);

}

**}**;

// determines successful run of a FA

template<typename FA>

relation **run**(lref<string> input

, lref<int> currSt=0) {

lref<char> firstCh;

lref<string> rest;

lref<int> nextSt;

return

eq(input,"") && FA::final(startSt)

|| head(input,firstCh) && tail(input,rest)

&& FA::transition(currSt,firstCh,nextSt)

&& recurse(run<FA>, rest, nextSt)

;

}

Given the above relations, any input string can be tested against the finite automata as follows:

if( run<Nfa>("aabba")**()** )

cout << "Matched";

else

cout << "No match";

## Query Expressions

…

# References

[CastorDesign] Roshan Naik, **Blending the Logic Paradigm into C++:** Discusses the design and implementation of the core of Castor that enables the LP in C++. http://www.mpprogramming.com/resources/CastorDesign.pdf

[CastorReference] Roshan Naik, **Castor** **reference manual:** http://www.mpprogramming.com/resources/CastorReference.pdf

[Leda] Timothy Budd, **Multiparadigm programming in Leda:** Addison-Wesley, 1995: A seminal piece of work in the field of multiparadigm programming. This book is a rich source of multiparadigm programming techniques.

**Acknowledgements:** Many thanks to Gardner Rust, Julius Gawlas, Ali Cehreli and Mansoor Peerbhoy for providing valuable suggestions on this document.

1. This is different from the approach taken in classical logic programming systems like Prolog. The general approach taken by Castor, of how to blend relations syntactically with the imperative languages, was pioneered by Timothy Budd in his multiparadigm language Leda. Although Castor shamelessly steals this idea from Leda, the underlying implementation techniques diverge significantly. [↑](#footnote-ref-1)
2. It may be interesting to note that the basic foundation to support logic programming is implemented in Castor in only about 400 lines of code. [↑](#footnote-ref-2)
3. This is same as the classification of pure and impure functions in Computer science. [↑](#footnote-ref-3)