Castor 1.1

Reference Manual

Roshan Naik

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

Last updated: Sep. 27, 09

**Table of Contents**

[1. Introduction 5](#_Toc241783384)

[2. Common Terms 6](#_Toc241783385)

[3. Core Facilities 7](#_Toc241783386)

[The Logic reference 7](#_Toc241783387)

[**Introduction** 7](#_Toc241783388)

[**Class Definition** 8](#_Toc241783389)

[**Construction, Copying and Destruction** 9](#_Toc241783390)

[**Assignment semantics** 10](#_Toc241783391)

[**Assignment** 10](#_Toc241783392)

[**Checked Access** 11](#_Toc241783393)

[**Unchecked Access** 11](#_Toc241783394)

[**Other Methods** 12](#_Toc241783395)

[**Non member functions** 12](#_Toc241783396)

[**Examples:** 12](#_Toc241783397)

[Type relation 13](#_Toc241783398)

[**Introduction** 13](#_Toc241783399)

[**Class Definition** 14](#_Toc241783400)

[**Construction** 14](#_Toc241783401)

[**Other Methods** 14](#_Toc241783402)

[**Examples:** 14](#_Toc241783403)

[Unification Support 15](#_Toc241783404)

[**Introduction** 15](#_Toc241783405)

[**eq relation** 16](#_Toc241783406)

[**eq\_f relation** 18](#_Toc241783407)

[**eq\_mf relation** 20](#_Toc241783408)

[**eq\_mem relation** 21](#_Toc241783409)

[**eq\_seq relation** 23](#_Toc241783410)

[Backtracking Support 24](#_Toc241783411)

[**Introduction** 24](#_Toc241783412)

[**Conjunction: Operator &&** 25](#_Toc241783413)

[**Inclusive Disjunction: Operator ||** 26](#_Toc241783414)

[**Exclusive Disjunction: Operator ^** 27](#_Toc241783415)

[Recursion 28](#_Toc241783416)

[**recurse relation** 28](#_Toc241783417)

[**recurse\_mf relation** 30](#_Toc241783418)

[Dynamic relations 31](#_Toc241783419)

[**Introduction** 31](#_Toc241783420)

[**Conjunctions relation** 31](#_Toc241783421)

[**Disjunctions relation** 32](#_Toc241783422)

[**ExDisjunctions relation** 33](#_Toc241783423)

[4. Inline Logic Reference Expressions (ILE) 36](#_Toc241783424)

[Operator overloads for creating ILEs: 36](#_Toc241783425)

[Named ILEs 38](#_Toc241783426)

[**call** 38](#_Toc241783427)

[**create** 40](#_Toc241783428)

[**Create::with**   **[deprecated, use create()]** 41](#_Toc241783429)

[**get** 42](#_Toc241783430)

[**mcall** 43](#_Toc241783431)

[**ref** 44](#_Toc241783432)

[Return types for ILE operators 45](#_Toc241783433)

[5. Higher-Order relations 47](#_Toc241783434)

[**zip relation** 47](#_Toc241783435)

[6. Utils 48](#_Toc241783436)

[Input/Output relations 48](#_Toc241783437)

[**read relation** 48](#_Toc241783438)

[**readFrom relation** 48](#_Toc241783439)

[**write relation** 49](#_Toc241783440)

[**writeTo relation** 50](#_Toc241783441)

[**write\_f relation** 51](#_Toc241783442)

[**writeTo\_f relation** 53](#_Toc241783443)

[**write\_mf relation** 54](#_Toc241783444)

[**writeTo\_mf relation** 55](#_Toc241783445)

[**write\_mem relation** 57](#_Toc241783446)

[**writeTo\_mem relation** 57](#_Toc241783447)

[**writeAll relation** 58](#_Toc241783448)

[**writeAllTo relation** 59](#_Toc241783449)

[Sequences and Containers 62](#_Toc241783450)

[**empty relation** 62](#_Toc241783451)

[**head relation** 62](#_Toc241783452)

[**head\_n relation** 63](#_Toc241783453)

[**head\_tail relation** 63](#_Toc241783454)

[**head\_n\_tail relation** 64](#_Toc241783455)

[**insert relation** 64](#_Toc241783456)

[**insert\_seq relation** 65](#_Toc241783457)

[**merge relation** 66](#_Toc241783458)

[**not\_empty relation** 67](#_Toc241783459)

[**sequence relation** 67](#_Toc241783460)

[**size relation** 70](#_Toc241783461)

[**tail relation** 71](#_Toc241783462)

[**tail\_n relation** 71](#_Toc241783463)

[Iterators 73](#_Toc241783464)

[**begin relation** 73](#_Toc241783465)

[**dereference relation** 73](#_Toc241783466)

[**end relation** 74](#_Toc241783467)

[**item relation** 74](#_Toc241783468)

[**next relation** 76](#_Toc241783469)

[**prev relation** 77](#_Toc241783470)

[Predicates 78](#_Toc241783471)

[**Boolean relation** 78](#_Toc241783472)

[**False relation** 78](#_Toc241783473)

[**True relation** 79](#_Toc241783474)

[**predicate relation** 79](#_Toc241783475)

[**predicate\_mf relation** 80](#_Toc241783476)

[**predicate\_mem relation** 82](#_Toc241783477)

[Other 83](#_Toc241783478)

[**dec relation** 83](#_Toc241783479)

[**defined relation** 83](#_Toc241783480)

[**inc relation** 84](#_Toc241783481)

[**defined relation** 85](#_Toc241783482)

[**error relation** 85](#_Toc241783483)

[**eval relation** 86](#_Toc241783484)

[**eval\_mf relation** 88](#_Toc241783485)

[**pause relation** 91](#_Toc241783486)

[**range relation** 92](#_Toc241783487)

[**repeat relation** 93](#_Toc241783488)

[**unique relation** 94](#_Toc241783489)

[**unique\_f relation** 94](#_Toc241783490)

[**unique\_mf relation** 95](#_Toc241783491)

[**unique\_mem relation** 96](#_Toc241783492)

[7. Cuts 98](#_Toc241783493)

[**Introduction** 98](#_Toc241783494)

[**cutexpr relation** 99](#_Toc241783495)

[**cut class** 100](#_Toc241783496)

[8. Helper classes, functions and macros 101](#_Toc241783497)

[**Coroutine class** 101](#_Toc241783498)

[**co\_begin macro** 104](#_Toc241783499)

[**co\_end macro** 104](#_Toc241783500)

[**co\_return macro** 104](#_Toc241783501)

[**co\_yield macro** 105](#_Toc241783502)

[**TestOnlyRelation class** 105](#_Toc241783503)

[**effective\_value function** 105](#_Toc241783504)

[**effective\_type class (meta function)** 106](#_Toc241783505)

[**getValueCont function** 107](#_Toc241783506)

[**OneSolutionRelation class [deprecated: use Coroutine]** 108](#_Toc241783507)

# Introduction

Castor is a pure C++ library that provides native support for the Logic paradigm (LP). Besides supporting LP, one of its key design goals is to allow easy mixing of LP with the other paradigms available in C++. Castor does not embed an interpreter or other logic programming engine to enable support for the logic paradigm. Instead it provides a few simple primitives which when put together enable LP. A discussion of the implementation techniques used in Castor to enable the Logic paradigm can found in the paper *“Blending the Logic Programming Paradigm into C++”*, available from http://[www.mpprogramming.com](http://www.mppprogramming.com).

This document serves only as a reference manual for Castor. For a tutorial on the Logic Paradigm and to get a better understanding on how to use Castor please refer to the paper *“**Introduction to Logic Programming in C++”*, also available from http://[www.mpprogramming.com](http://www.mppprogramming.com).

Castor is a pure header library and does not require your applications to link with any additional static or shared libraries other than the standard C++ library. It does not require any language extensions or special preprocessing to enable LP. All facilities are part of the castor namespace. Including the header file *castor.h*, makes this namespace and all castor facilities available for use. The following is a trivial hello world program using Castor:

#include "castor.h"

using namespace castor;

int main() {

write(“Hello World”)();

return 0;

}

# Common Terms

**Logic reference:** Variable of type lref<T>.

**Plain old type (POT):** All types other than lref<T>.

**Effective type**: Effective type of a logic reference lref<T1> is T1. Effective type of any other type X is X itself.

**Effective value:** If t1 is a logic reference then its effective value is obtained by the expression \*t1. Effective value of any other object t2 is t2 itself. The effective value of a logic reference is also known as the *referenced object*.

**Relation**: Typically refers to a function or member function having return type relation (or a type convertible to relation). Sometimes it may also refer to objects of type relation (or a type convertible to relation). The distinction, if needed, is usually inferred from the context in which the term is used.

# Core Facilities

## The Logic reference

**Introduction**

Template type lref, abbreviation for *logic reference*, provides a facility for passing values in/out of relations in the form of arguments. It is essentially a reference counted smart pointer designed to realize logic and functional programming techniques. It is not intended to substitute general purpose smart pointers (such as std::auto\_ptr or boost::shared\_ptr) which are primarily designed with the intent of simplifying memory management. The object referenced by a logic reference is called the *referenced object*.

A logic reference always refers to a copy of the value assigned to it. This copy is kept on the heap and can be accessed by dereferencing the lref. Initializing an lref<T> with another lref<T> (i.e. copy construction), causes both logic references to be *bound* *together*. Bound lrefs refer to the same object. Thus any change to the referenced object is observed by all lrefs bound to it.

Lrefs can only be bound by initialization (i.e. copy construction) and not by assignment. A binding between lrefs cannot be broken. The referenced object is deallocated by the destructor of the last lref referencing it. An lref that is default constructed does not refer to any object unless a value is assigned to it. Lrefs that do not reference anything are said to be *undefined* or *uninitialized*. An initialized lref may be uninitialized by invoking the reset method. Resetting an lref will implicitly cause all lrefs bound with it to also be undefined. Resetting does not deallocate the referenced object.

Figure 1 below demonstrates the internal structure for the following logic references:

lret<int> lref1 = 5;

lret<int> lref2 = lref1;

lret<int> lref3;

lref1

lref2

lref3

(useCount=2)

(defined=true)

**5**

(useCount=1)

(defined=false)

referenced obj

shared pointer

shared pointer

Figure . Internal structure of Logic References

Here lref1 and lref2 are bound together. They are also *defined*, as their shared pointer actually refers to an object. Since lref3’s shared pointer does not refer to any object, lref3 is *undefined*.

As noted above, whenever a value is assigned to a logic reference it maintains of copy the assigned value. The lifetime of this object is then managed by the logic reference. There can be situations when we may want to have an lref pointing to a particular object (and not its copy). This enables us grab objects that emanate from anywhere in the system and treat them relationally using an lref. Sometimes operating on a copy may not be practical, too expensive or even plain wrong.

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

**Class Definition**

// requires: T should support copy construction and copy assignment

template <typename T>

class lref {

public:

typedef T result\_type;

// Construct/Copy/Destroy

lref();

lref(const T& value);

template<typename T2> // requires: T provides T::T(const T2&)

lref(const T2& value);

lref(const lref<T>& rhs);

private:

template<typename T2>

lref(const lref<T2>& rhs);

public:

lref(T\* ptr, bool manage);

~lref();

// Assignment

lref& operator =(const T& newValue);

template<typename T2>

lref& operator=(const T2& newValue);

lref& operator=(const lref& rhs);

template<typename T2>

lref& operator=(const lref<T2>& rhs);

template<typename T2>

void set\_ptr(T2\* ptr, bool manage);

// Checked access

T& operator \*();

const T& operator \*();

..*unspecified*.. operator ->();

const .. & operator ->() const ;

// Unchecked access

T& get();

const T& get() const;

// Other

void reset(); // nothrow

bool defined() const; // nothrow

unsigned int use\_count() const; // nothrow

void swap(lref<T>& other); // nothrow

}; // class lref

// Disable template instantiation of lref<T&> and lref<void>

template<> class lref<void>;

template<typename T> class lref<T&>;

// Non member swap(). Calls l.swap(r)

template<typename T>

void swap(lref<T>& l, lref<T>& r);

**Construction, Copying and Destruction**

lref()

Constructs a logic reference that does not refer to any object. An lref that does not refer to any object is said to be *undefined* or *uninitialized*. On completion, reference count is set to 1.

lref(const T& value)

Constructs a logic reference that refers to a copy of value on the heap. The referenced object is instantiated using the expression new T(value). Lifetime of this referenced object will be managed by the lref. On completion reference count is set to 1.

template<typename T2> lref(const T2& value)

Constructs an lref<T>. The referenced object is instantiated using the expression :

* new T2(value), if T2 is publicly derived from T. *OR*
* new T(value), if T is copy constructible from T2 (and *not* derived from T).

If T2 is not derived from T, a converting constructor T::T(const T2&) must be available. Note that this overload is invoked only when newValue is *not* exactly of type T. Lifetime of the referenced object will be managed by the lref. On completion, reference count is set to 1.

lref(const lref<T>& rhs)

Constructs a logic reference that refers to the same object that is referenced by rhs. On completion, both lrefs will also share the same reference count which will be incremented by 1. The two logic references are now *bound together*.

template<typename T2> lref(const lref<T2>& rhs)

This is a private constructor. It disables construction of lref<T> from lref<T2>.

lref(T\* ptr, bool manage)

Constructs a logic reference that refers to the same object as \*ptr. The lifetime of the object referenced by ptr will be managed if manage is true. On completion reference count is set to 1.

~lref();

On completion, reference count is decremented by 1. If the reference count has reached 0, and the referenced object (if any) is being managed, the referenced object will be deleted.

***Assignment semantics***

From the standpoint of implementing assignment to an lref, it is possible to update the referenced object with the new object in one of two ways:

1. A simple assignment of newValue to the currently referenced object, OR
2. First deallocate the currently referenced object (if any), then allocate a new object initialized with newValue to replace the old object.

The first strategy is typically more efficient since it does not involve allocation and de-allocation of the referenced object. However it is not always feasible to use it. For optimization reasons, no guarantees are provided as to which of the above may actually occur. The strategy used typically depends on the types of the currently referenced object and newValue. For instance, the current implementation uses the following strategy to minimize calls to new and delete:

1. Use simple assignment if lref<T> is defined and the referenced object and newValue are both exactly of type T.
2. Otherwise, the referenced object, if any, is deallocated and replaced with a new object initialized with newValue. If newValue is of a type T2 such that T2 is derived from T, the new object is allocated using the expression new T2(newValue). If newValue is of type T2, such that T is copy constructible from T2 (and not derived from T) then the new object is allocated using the expression new T(newValue).

**Assignment**

lref& operator=(const T& newValue);

Assigns newValue to the logic reference. Note that this overload is invoked only when newValue is exactly of type T (and not for types derived from or copy constructible from T).

template<typename T2>

lref& operator=(const T2& newValue);

Assigns newValue to the logic reference. Note that this overload is invoked only when T2 is publicly derived from T.

lref& operator=(const lref<T>& rhs);

Assigns rhs’s referenced object to this logic reference. This operation does *not* cause the two logic references to be bound together. If rhs is not initialized, this lref will also be reset. Note that this overload is invoked only when the type of rhs is the same as this lref.

template<typename T2>

lref& operator=(const lref<T2>& rhs);

Assigns rhs’s referenced object to this logic reference. This operation does *not* cause the two logic references to be bound together. If rhs is not initialized, this lref will also be reset. Note that this overload is invoked only if T2 is *not* the same T1 and T2 is assignable to T.

template<typename T2>

void set\_ptr(T2\* ptr, bool manage)

Causes the lref to refer to the object pointed to by ptr. Unlike other forms of assignment, this operation does *not* make a copy of the object pointed to be ptr. If the lref currently references an object that whose lifetime if managed, the object will be deleted. The lifetime of the object referenced by ptr will be managed if manage is true. Note that T2 should either be same as T, or T2\* should be assignable to T\*.

**Checked Access**

T& operator \*();

Returns the referenced object. If lref is undefined, this operation throws an exception of type InvalidDeref.

const T& operator \*();

Returns the referenced object. Throws InvalidDeref if the lref is undefined.

..*unspecifiedType*.. & operator ->();

This method is used to access the members of the referenced object. Exact return type is deliberately unspecified. Throws InvalidDeref if the lref is undefined.

const .. & operator ->() const ;

This method is used to access the members of the referenced object. Exact return type is deliberately unspecified. Throws an exception of type InvalidDeref if the lref is undefined.

**Unchecked Access**

T& get();

Returns the referenced object. If the lref is not initialized, its behavior is undefined. This method is the unchecked equivalent of operator \*.

const T& get() const;

Returns the referenced object. If the lref is not initialized, its behavior is undefined. This method is the unchecked equivalent of operator \*.

**Other Methods**

void reset(); // nothrow

This method causes the lref to be undefined. The referenced object (if any) will not be deallocated by this operation.

bool defined() const; // nothrow

Checks if the lref is currently defined.

unsigned int use\_count() const; // nothrow

Returns the total number of logic references that are bound with this lref. This value is always greater than or equal to 1.

void swap(lref<T>& other); // nothrow

Swaps the pointer to the referenced object stored in the shared pointers of the two lrefs. Use count (i.e. reference count) is not swapped.

**Non member functions**

template<typename T>

void swap(lref<T>& l, lref<T>& r);

Swaps the referenced objects of l and r. Semantics are same as l.swap(r).

**Examples:**

// Accessing referenced object

string s="logic";

lref<string> ls1 = s, ls2 = "paradigm";

cout << \*ls1 << " "; // checked access

cout << ls2.get() << "\n"; // unchecked access

// Behavior of bound lrefs

lref<string> ls3 (ls1); // ls3 and ls1 are now bound

ls1 = "multi";

cout << \*ls3; // prints "multi"

cout << std::boolapha << ls1.defined(); // prints "true"

ls3.reset();

cout << std::boolapha << ls1.defined(); // prints "false"

ls3 = ls2; // this does not bind the two lrefs

ls2.reset();

cout << std::boolapha << ls1.defined(); // prints "true"

// Swapping lrefs

lref<int> li1 = 2;

lref<int> li2(li1); // bind

lref<int> li3;

swap(li2,li3);

cout << \*li3 << "\n"; // prints "2"

// foll. prints: "false 2"

cout << boolalpha << li2.defined() << " " << li2.use\_count() << "\n";

// foll. prints: "true 1"

cout << boolalpha << li3.defined() << " " << li3.use\_count() << "\n";

## Type relation

**Introduction**

The concept of a relation is to the logic paradigm what a function is to the imperative paradigm. Relations are the basic computational building blocks when programming in logic. Since C++ is based on the imperative paradigm, it is desirable to be able to describe relations as functions. Fortunately, due to the flexibility of C++, this is possible to do without extending the language. This allows relations to be given similar treatment as regular functions, facilitating the logic paradigm to blend smoothly into C++. Regular functions can be composed from other functions and relations. Similarly relations can be composed from other relations and functions[[1]](#footnote-1). The type relation enables this kind of integration with bare minimal syntactic overhead.

The type relation is typically used to specify the return type for functions and methods that represent the concept of a relation. Functions and methods with return type relation are themselves referred to as relations. In this manual we refer to such a function or method as a “relation” and to the type as “relation”. (Note the difference in fonts used). The term “relation” is commonly (i.e. outside of this reference manual) used to refer to the former. The following are a few examples of relations defined using the type relation:

relation **twiceOf**(lref<int> x, lref<int> x2) { // non member relation

…

}

struct Arithmetic {

relation **twiceOf**(lref<int> x, lref<int> x2) { // member relation

…

}

};

From an operational semantics point of view, relation is used to hold function objects with return type bool and no arguments. A relation internally stores a copy of such a function object for delayed invocation. Application of the function call operator, without any arguments, on a relation object triggers the invocation of the stored function object. An object of type relation cannot be default initialized (i.e. without arguments). This ensures that a relation is always initialized with some function object and thus it is always safe to apply the function call operator on a relation. A different function object can be assigned to an instance of relation after initialization.

**Class Definition**

class relation {

public:

typedef bool result\_type;

// Requires : F supports method... bool F::operator()(void)

template<class F>

relation(F f);

relation(const relation& rhs);

relation& operator=(const relation& rhs);

bool operator()(void) const;

};

**Construction**

template<class F>

relation(F f);

Constructs a relation from function object f. F is expected to support bool F::operator()(void). Note, F must be a non-static member function type.

relation(const relation& rhs);

Copy constructs relation from another. This involves storing a copy of the function object stored in rhs.

**Other Methods**

relation& operator=(const relation& rhs);

Copies rhs into this relation.

bool operator()(void) const;

Triggers evaluation of the internally stored function object.

**Examples:**

struct PrintHello {

bool operator() (void) {

cout << "Hello ";

return true;

}

};

struct PrintWorld {

bool operator() (void) {

cout << "World";

return true;

}

};

relation r = PrintHello();

r(); // invokes PrintHello::operator()

r = PrintHello() && PrintWorld();

r(); // invokes PrintHello::operator() then PrintWorld::operator()

relation r2; // Compiler Error! relation canot be default initialized.

## Unification Support

**Introduction**

Logic paradigm uses a general purpose problem solving technique for evaluating relations to perform computation. This technique involves two fundamental operations: unification and backtracking. In a nutshell, these two operations can be described as follows:

* Backtracking determines which path of evaluation should be pursued next from a set of (possibly empty) available paths.
* Unification either produces results or tests if a desired result was produced.

This section only covers unification and the facilities provided in Castor to support unification. Backtracking is covered in the next section. The unification operation is simply an *attempt* to unify values of two items. The items could be logic references or plain old types (i.e. types other than lref). To unify two objects means to make their values equal. The definition of equality is governed by operator ==. When attempting to unify two objects, it may be the case that the two objects compare equally. In such a case unification succeeds trivially. Assignment of one object to another is considered only if one of the two objects is an uninitialized logic reference. The uninitialized logic reference is assigned the value of the other object thus making the two objects equal. In all other cases unification fails. Relation eq provides the fundamental support for unification. The semantics of unification relation eq is as follows:

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

Castor provides a few unification relations. The choice of which relation to use primarily depends upon the nature of items being unified. The most basic unification support is provided by relation eq. Any two objects that can be compared and assigned to each other can be unified using eq. Unification of containers such a std::list with a sequence of values bounded by an iterator pair is supported by eq\_seq. A more sophisticated facility for unification of sequences with values, iterator pairs or other sequences is provided by relation sequence. Relations eq\_f and eq\_mf provide support for unification of objects with values returned from functions and member functions respectively.

**eq relation**

//1. Unify logic references

template<typename L, typename R>

UnifyLR<L,R> **eq**(lref<L>& l, lref<R>& r)

template<typename L, typename R, typename Cmp>

UnifyLR<L,R,Cmp> **eq**(lref<L>& l, lref<R>& r, Cmp cmp)

//--- Treat char\* as strings instead of pointer to a char ---

//2. Unify logic reference with char\*. Used when T is an abstraction for char\* (like std::string)

template<typename T>

UnifyL<T,T> **eq**(const lref<T>& l, const char\* r)

template<typename T, typename Cmp>

UnifyL<T,T,Cmp> **eq**(const lref<T>& l, const char\* r, Cmp cmp)

template<typename T>

UnifyL<T,T> **eq**(const char\* l, const lref<T>& r)

template<typename T, typename Cmp>

UnifyL<T,T,Cmp> **eq**(const char\* l, const lref<T>& r, Cmp cmp)

//3. Unify two char\* strings

Boolean **eq**(const char\* l, const char\* r)

template<typename Cmp>

Boolean **eq**(const char\* l, const char\* r, Cmp cmp)

//--- Remaining overloads provided for optimization ---

//4. Neither argument is a logic reference

template<typename L, typename R>

Boolean **eq**(const L& l, const R& r)

template<typename L, typename R, typename Cmp>

Boolean **eq**(const L& l, const R& r, Cmp cmp)

//5. one argument is a logic reference but the other is not

template<typename L, typename R>

UnifyL<L,R> **eq**(const lref<L>& l, const R& r)

template<typename L, typename R, typename Cmp>

UnifyL<L,R,Cmp> **eq**(const lref<L>& l, const R& r, Cmp cmp)

template<typename L, typename R>

UnifyL<R,L> **eq**(const L& l, const lref<R>& r)

template<typename L, typename R, typename Cmp>

UnifyL<R,L,Cmp> **eq**(const L& l, const lref<R>& r, Cmp cmp)

**Declarative reading**: l is equal to r.

**Template Parameters:**

L,R,T : Should satisfy the standard *CopyConstructible* [$20.1.3]*, Assignable* [$23.1.4] and *EqualityComparable* [$20.1.1] requirements.

Cmp : A function or function object type which accepts two arguments of type T. Used to customize the comparison operation performed during unification.

**Parameters:**

l : [in/out] Item to be unified with r.

r : [in/out] Item to be unified with l.

cmp : [in] Binary predicate used to compare two objects of type T.

If both l and r are logic references, at least one of them must be initialized at the time of evaluation.

**Exceptions:**

InvalidDeref : If both l and r are not initialized at the time of evaluation.

**Notes:**

Relation eq will either check or make the two arguments equal. If both arguments are initialized to a value, then eq will compare them for equality and succeeds if the two are equal and fails (i.e. returns false) otherwise. If one of the arguments is *not* initialized, then the value of the other argument is assigned to it, thus making the two arguments equal. If both arguments are not initialized at the time of evaluation, an exception is thrown.

This operation of testing/assigning depending upon the whether or not the two arguments are initialized, is referred to as **unification**. Unification, in a sense, is the relational equivalent of operator== (which only performs a test for equality) and operator= (which only performs assignment). In short, relation eq unifies its arguments.

**Examples:**

// 1) with simple values and value types(i.e. not logic references)

eq(2,2)**()**; // compare 2 with 2 .. returns true

eq(1,2)**()**; // compare 1 with 2 .. returns false

int i=2;

eq(i,2)**()**; // compare value of i with 2 .. returns true

// 2) with initialized logic references

lref<int> li=2;

eq(i,li)**()**; // compare i with li .. returns true

// 3) with uninitialized logic references

lref<int> lj; // note: lj is not initialized with a value

eq(lj,i)**()**; // lj is assigned value of i, thus initializing lj

cout<< \*lj; // prints "2"

lref<int> lk; // at this point lk is not initialized but lj is

eq(lj,lk)**()**; // lk is assigned value of lj, thus initializing lk

lj.reset(); // uninitialize lj

lj.reset(); // uninitialize lk

eq(lj,lk)**()**; // throws InvalidDeref

// 4) unifying containers

lref<vector<int> > lvi;

vector<int> vi = /\* 1,2,3,4 \*/;

eq(lvi,vi)**()**; // lvi is assigned a vector equivalent to vi

**Also refer to:**

sequence, eq\_seq, eq\_f, eq\_mf

**eq\_f relation**

// overloads for functions objects

template<typename T, typename Func>

Eq\_f\_r<T, Func>

**eq\_f**(const lref<T> l, Func f)

template<typename T, typename Func1, typename A1>

Eq\_f\_r1<T, Func1, A1>

**eq\_f**(lref<T> l, Func1 f, const A1& a1\_)

template<typename T, typename Func2, typename A1, typename A2>

Eq\_f\_r2<T, Func2, A1, A2>

**eq\_f**(lref<T> l, Func2 f, const A1& a1\_, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to f

// overloads for function pointers

template<typename T, typename R>

Eq\_f\_r<T,R(\*)(void)>

**eq\_f**(lref<T> l, R(\* f)(void))

template<typename T, typename R, typename P1, typename A1>

Eq\_f\_r1<T,R(\*)(P1),A1>

**eq\_f**(lref<T> l, R(\* f)(P1), const A1& a1\_)

template<typename T, typename R, typename P1, typename P2, typename A1

, typename A2>

Eq\_f\_r2<T,R(\*)(P1,P2),A1,A2>

**eq\_f**(lref<T> l, R(\* f)(P1,P2), const A1& a1\_, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to f

**Declarative reading**: l is equal to the value returned by invoking f( a1\_,..,aN\_).

**Template Parameters:**

T : Any type which satisfies requirements of logic reference.

Func*N* : A function object type with arity *N*.

R: Return type of the function pointer.

P*n*: Type of the Nth parameter of function pointer. Can be an lref or POT. A*N* should be either same as or convertible to the corresponding P*n*.

A*n* : Type of argument passed at position *n* to the Func*N* type. Can be a POT or lref whose effective type is convertible to the corresponding parameter type in Func*N*.

**Parameters:**

l : [in/out] Item to be unified with result of f.

f : [in] Function pointer or function object whose result will be unified with l. Cannot be a member function type.

a*N\_* : [in] Argument (POT or lref) at position *N* whose effective value will be passed to f.

**Exceptions:**

InvalidDeref : If any argument aN is an lref and is not initialized at the time of evaluation.

Any exception thrown by f.

**Notes:**

Relation eq\_f provides support for unification with values returned by evaluating functions or function objects. ILEs may also be used as arguments to parameter f. Parameter l will be compared with or assigned the result of evaluating f. All arguments (if any) required to evaluate f should be passed to eq\_f using the a*N\_* parameters. Effective value of every a*N\_* will be passed to f.

**Examples:**

// 1) With regular functions

int compute(int j, int k) {

return j/k-1;

}

lref<int> li, lj, lk;

relation r = eq(lj,6) && eq(lk,2) && eq\_f(li, &compute, lj, lk);

if(r**()**)

cout << \*li; // prints "2"

// 2) With function objects

struct Compute {

int operator ()(int j, int k) {

return j/k-1;

}

};

lref<int> li, lj, lk;

relation r = eq(lj,6) && eq(lk,2) && eq\_f(li, Compute(),lj,lk);

if(r**()**)

cout << \*li; // prints "2"

// 3) With ILE (Inline Lref Expression)

lref<int> li, lj, lk;

relation r = eq(lj,6) && eq(lk,2) && eq\_f(li, lj/lk-1 );

if(r**()**)

cout << \*li; // prints "2"

**Also refer to:**

eq\_mf, eq, eq\_seq

**eq\_mf relation**

// Overloads for non-const member functions

template<typename L, typename R, typename Obj>

Eq\_mf\_r0<L,Obj,R(Obj::\*)(void)>

**eq\_mf**(lref<L> l, lref<Obj>& obj\_, R(Obj::\*mf)(void) )

template<typename L, typename R, typename P1, typename Obj

, typename A1>

Eq\_mf\_r1<L,Obj,R(Obj::\*)(P1),A1>

**eq\_mf**(lref<L> l, lref<Obj>& obj\_, R(Obj::\* mf)(P1), const A1& a1\_)

template<typename L, typename R, typename P1, typename P2

, typename Obj, typename A1, typename A2>

Eq\_mf\_r2<L,Obj,R(Obj::\*)(P1,P2),A1,A2>

**eq\_mf**(lref<L> l, lref<Obj>& obj\_, R(Obj::\* mf)(P1,P2), const A1& a1\_

, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to mf

// Overloads for const member functions

template<typename L, typename R, typename Obj>

Eq\_mf\_r0<L,Obj,R(Obj::\*)(void) const>

**eq\_mf**(lref<L> l, lref<Obj>& obj\_, R(Obj::\*mf)(void) const)

template<typename L, typename R, typename P1, typename Obj

, typename A1>

Eq\_mf\_r1<L,Obj,R(Obj::\*)(P1) const,A1>

**eq\_mf**(lref<L> l, lref<Obj>& obj\_, R(Obj::\* mf)(P1) const

, const A1& a1\_)

template<typename L, typename R, typename P1, typename P2

, typename Obj, typename A1, typename A2>

Eq\_mf\_r2<L,Obj,R(Obj::\*)(P1,P2) const,A1,A2>

**eq\_mf**(lref<L> l, lref<Obj>& obj\_, R(Obj::\* mf)(P1,P2) const

, const A1& a1\_, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to mf

**Declarative reading**: l is equal to the value returned by invoking member function mf on object obj\_ with arguments p1..pN.

**Template Parameters:**

L : Any type which satisfies requirements of logic reference.

Obj : A type whose member function is to be invoked.

R : Return type of the member function.

P*n*: Type of the nth parameter of member function.

A*n* : Type of the nth argument to being passed. Can be a POT or lref whose effective type is convertible to the corresponding parameter type P*n*.

**Parameters:**

l : [in/out] Item to be unified with result of evaluating member function mf on obj\_.

obj\_ : [in] Object on which member function pointed to by mf will be invoked. This argument must be a logic reference. This restriction ensures methods are invoked on the actual argument and not on a copy of obj\_.

mf : Member function pointer whose result is to be unified with l.

a*N\_* : [in] Argument (POT or lref) at position *N* whose effective value will be passed to mf.

**Notes:**

Relation eq\_mf provides support for unification with values returned by evaluating member functions on objects. Parameter l will be compared with or assigned the result of evaluating obj\_->\*mf(..).Note that any side effects induced by mf will not be undone during backtracking. Hence eq\_mf should be used with care, ensuring that it does not interfere with the correct evaluation of other relations by modifying obj\_ or other objects that are shared with other relations. eq\_mf always succeeds at most once.

**Examples:**

struct Compute {

int j;

Compute(int j) : j(j)

{}

int apply(int k) { // unary member function to be invoked

return j/k-1;

}

};

lref<int> li;

lref<Compute> comp = Compute(6);

relation r = eq\_mf(li, comp, &Compute::apply, 2);

if(r**()**)

cout << \*li; // prints "2"

**Also refer to:**

eq\_f, eq, eq\_seq, eval\_f, eval\_mf

**eq\_mem relation**

template<typename L, typename Obj, typename MemberT>

Eq\_mem\_r<L, Obj, MemberT>

**eq\_mem**(lref<L> l, lref<Obj>& obj\_, MemberT Obj::\* mem)

**Declarative reading**: l is equal to member variable Obj::mem.

**Template Parameters:**

L : Any type which satisfies requirements of logic reference.

Obj : Any type which whose member variable is to be accessed.

MemberT : Type of the data member to be accessed.

**Parameters:**

l : [in/out] Item to be unified with result of evaluating member function mf on obj\_.

obj\_ : [in] The object whose data member is to be accessed. This argument must be a logic reference. This restriction ensures methods are invoked on the actual argument and not on a copy of obj\_.

mem : Pointer to a member variable which is to be unified with l.

**Notes:**

Relation eq\_mem provides support for unification with member variables. Parameter l will be compared with or assigned the result of evaluating (\*obj\_).\*mem.

**Examples:**

// Get a first item in a pair of strings representing a person’s name

lref<pair<string,string> > p = pair<string,string>("Roshan","Naik");

lref<string> firstName;

eq\_mem(firstName, p, &pair<string,string>::first)**()**;

cout << \*firstName; // prints "Roshan"

// Compute total salary of all employees

struct employee {

string name;

int salary;

bool operator == (const employee& rhs) const {

return name==rhs.name && salary==rhs.salary;

}

};

list<employee> employees = ...;

lref<int> salary;

lref<employee> e;

relation salaries = item(e, employees.begin(), employees.end())

&& eq\_mem(salary, e, &employee::salary);

int total=0;

while(salaries())

total+=\*salary;

cout << total;

**Also refer to:**

eq, eq\_f, eq\_mf, eq\_seq

**eq\_seq relation**

template<typename Cont, typename Iter>

UnifySeq<Cont,Iter> **eq\_seq**(const lref<Cont>& c, Iter begin\_, Iter end\_)

template<typename Cont, typename Iter, typename Cmp>

UnifySeq<Cont, Iter, Cmp> **eq\_seq**(const lref<Cont>& c, Iter begin\_, Iter end\_, Cmp cmp)

**Declarative reading**: Container c is equal to the sequence represented by the iterators begin and end.

**Template Parameters:**

Cont : Must satisfy requirements of standard C++ containers [$23.1].

Iter: A type that yields Cont::value\_type on deferencing and supports postfix increment and decrement operators[$23.1]. In other words, any iterator or pointer type. Iter type is not limited the Cont::iterator. Thus it is possible for Iter to be, for example, the type list<T>::iterator or T\* when Cont is the type vector<T>.

Cmp : A binary predicate with both parameters of type T such that Cont::value\_type can be converted to T. Used to customize the comparison operation performed during unification.

**Parameters:**

c : [in/out] Item to be unified with r.

begin\_ : [in] Iterator to the beginning of the sequence to be unified with c.

end\_ : [in] Iterator to one past the end of the sequence to be unified with c.

cmp : [in] cmp : [in] Binary predicate used to compare two objects of type Cont::value\_type. It may be a function object or pointer to function. cmp is used to compare an item in container c with the corresponding item in sequence[begin\_,end\_)for equality. cmp cannot be a logic reference.

**Exceptions:**

InvalidDeref : If either begin\_ or end\_ is not initialized at the time of evaluation.

**Notes:**

Relation eq\_seq provides a simple and useful facility for unifying containers with a sequences represented by an iterator pairs. If iterators begin\_ and end\_ are logic references they must be initialized at the time of evaluation. If c is initialized, then the sequence [ c.begin(), c.end() ) is compared for equality with the sequence [begin\_, end\_). Comparison fails if the two sequences differ in length or if the items in the corresponding positions do not compare equally. Comparison operation can be customized by passing a binary predicate to parameter cmp. If c is not initialized, it will be initialized with a container consisting of the elements in [begin\_,end\_).

**Examples:**

// 1) generate container with elements

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

lref<vector<int> > vi;

if( eq\_seq(vi, ai, ai+3)**()** )

copy(vi->begin(), vi->end(), ostream\_iterator<int>(cout, " "));

// 2) compare container with sequence

list<int> li = /\* {1,2,3} \*/;

lref<vector<int> > vi = vector<int>(3);

(\*vi)[0]=1; (\*vi)[1]=2; (\*vi)[2]=3;

if( eq\_seq(vi, li.begin(), li.end())**() )**

cout << "vi has the same elements as li";

**Also refer to:**

sequence, eq

## Backtracking Support

**Introduction**

As in life, our main purpose in computation is to find *the* answer. There are often many ways to get to an answer. Some problems may have one or more answers, and other problems may have none. The search for an answer can possibly lead in several different directions, not all of which are actually productive. The search must therefore be able to step back and resume the search from an earlier point where an alternative direction was possible. This process pursuing one path of evaluation and then stepping back to try an alternative in search of a solution is referred to as backtracking.

Consider the relational expression eq(x,3) && eq(y,4) || eq(x,5) && eq(y,6) which basically declares “if x is 3 then y is 4, OR if x is 5 then y is 6”. We can represent this using the following expression tree:

||

&&

eq(x,3)

eq(y,4)

&&

eq(x,5)

eq(y,6)

Expression tree for : eq(x,3) && eq(y,4) || eq(x,5) && eq(y,6)

Let us see how backtracking goes about answering the question “What is a suitable value for x given that y’s value is fixed to 6?”. Goal of backtracking is to evaluate this expression tree successfully. Evaluation begins at the top of the tree and encounters the disjunction operator || which offers two possible choices (the left and right branches) in order to proceed. For disjunction to succeed, it is sufficient that any one branch evaluates successfully. Backtracking relies on depth first strategy and chooses the left branch. Here the conjunction operator && is encountered. For the conjunction to succeed, both its branches must evaluate successfully. The left branch is evaluated first followed by the right branch. The left branch succeeds in unifying x with 3 (since x is not initialized) but the right branch fails to unify y with 4 (since y is initialized to 6). This leads to the failure of the conjunction which in turn implies failure of the entire left branch under the disjunction operator.

Now comes the time to step back and resume exploring the right branch of the disjunction. Stepping back involves reverting any side effects that occurred while pursuing the left branch under disjunction. In this case x went from being uninitialized to being initialized with 3. So X is reset to its uninitialized state. The evaluation now steps down the right branch in a fashion same as before and attempts to unify x with 5 and y with 6. This time around both operations succeed and consequently the conjunction also succeeds. This in turn implies successful evaluation of the disjunction and the entire expression tree. We now find x initialized with 5 which answers the question we started out to with.

As can be seen from the above example, conjunction and disjunction operators are used to build the expression tree that is traversed during backtracking. Castor provides native support for two varieties of disjunction: inclusive and exclusive. Typically, classic logic programming systems such as Prolog directly support only the inclusive variety. Operator ||, which is used in the above example, provides support for inclusive disjunction. Support for exclusive disjunction is provided by the ex-or operator ^. The following sections deal with each of the relational operators in further detail.

**Conjunction: Operator &&**

And\_r<relation,relation> operator && (const relation& l

, const relation& r)

Logical conjunction is a binary relation between any two relations with a meaning similar to “and” in English. Conjunction is denoted by operator && in Castor. It is itself a relation which takes two other relations as arguments. In other words, it is a higher order binary relation. A simplified definition for conjunction is: a relation that succeeds when both its argument relations succeed. However, in logic, a relation may succeed zero, one or more times. The definition of conjunction needs to accommodate possibly multiple successful evaluations of its argument relations. Thus we broaden the definition of conjunction to: a relation that succeeds each time the second relation evaluates successfully for a successful evaluation of the first. Given this definition, conjunction can itself succeed zero, one or more times depending upon its argument relations. For instance consider the following simple expression:

range(x, 1, 3) && range(x, 2, 5)

Both arguments to && are range relations. The first relation indicates that x is a value in the inclusive range 1 through 3. The second relation indicates that x is a value in the inclusive range 2 through 5. Considered in isolation, the first relation can succeed three times (producing values 1,2 and 3) and the second relation can succeed four times (producing values 2,3,4 and 5). The conjunction itself can succeed only twice, i.e. when x is 2 or 3. Any other value for x will fail either the first or the second range relation.

Operational semantics of && can be summarized as: pursue all solutions in r for each solution in l. The following pseudo code demonstrates this.

//This is psuedo code!

relation tmp = rhs; //make copy of rhs

while( l() ) {

while( r() )

**yield** return true;// Not C++. ‘yield’ keyword borrowed from C#

rhs = tmp; //reset rhs

}

//no more solutions left

return false;

In terms of the imperative paradigm, the operational semantics can be visualized as one loop nested in another as shown above. The outer loop is responsible for evaluating l and the inner loop for evaluating r. If l succeeds, r is attempted. If r succeeds the conjunction also succeeds and returns true. Further attempts to pursue more solutions from conjunction will lead to repeated evaluations of r until it fails. Once r is exhausted, l is evaluated once again and the whole process repeats. Finally when l is exhausted the conjunction has no more solutions to produce and returns false. Thereafter any attempts to pursue more solutions from this conjunction will fail immediately.

Note that once all solutions are exhausted in r, it has to be reset to its original state before pursuing the next solution in l. This enables r to resume producing a new set of solutions for each successful evaluation of l. Otherwise r will not be able to produce any new solutions. In order to be able to reset r to its original state, a copy of r is made in tmp prior to attempting any evaluation of r.

**Inclusive Disjunction: Operator ||**

// requires : L and R must be treatable as relations

template<typename L, typename R>

Or\_r<L,R> operator || (const L& l, const R & r)

Inclusive disjunction is a binary relation between any two relations with a meaning similar to “or” in English. It is denoted by operator || in Castor. Inclusive disjunction is itself a relation which takes two other relations as arguments. In other words, it is a higher order binary relation. A simplified definition for inclusive disjunction is: a relation that succeeds when at least one of its argument relations succeeds. To accommodate the ability of its argument relations to evaluate successfully zero or more times, we broaden its definition to: a relation that succeeds each time there is at a successful evaluation of one of its argument relations. Given this definition, disjunction can itself succeed zero, one or more times depending upon its argument relations. For instance consider the following simple expression:

range(x, 1, 3) || range(x, 2, 5)

Both arguments to || are range relations. The first relation indicates that x is a value in the inclusive range 1 through 3. The second relation indicates that x is a value in the inclusive range 2 through 5. Considered in isolation, the first relation can succeed three times (producing values 1,2 and 3) and the second relation can succeed four times (producing values 2,3,4 and 5). The combined expression itself can succeed seven times (producing values 1,2,3,2,3,4 and 5). Relation range(x,1,3) produces the first three values for x and range(x,2,5) produces the remaining four values. Notice how the duplicate values are generated for x.

Operational semantics of operator || can be summarized as: pursue all solutions in l then pursue all solutions in r. The following pseudo code demonstrates this.

//This is psuedo code!

while( l() )

**yield** return true; // Not C++. ‘yield’ keyword borrowed from C#

while( r() )

**yield** return true;

return false; //no more solutions left

In terms of the imperative paradigm, the operational semantics can be visualized as two main loops, one following another as shown above. The first loop is responsible for evaluating l and the second for evaluating r. After l’s solutions have been exhausted by the first loop, the second loop pursues solutions in r. Once r has also been exhausted, there are no more solutions to produce and evaluation enters the third loop. Here disjunction always fails by returning false. Thus all future attempts to pursue more solutions from this disjunction will fail immediately.

**Exclusive Disjunction: Operator ^**

// requires : L and R must be treatable as relations

template<typename L, typename R>

ExOr\_r<L,R> operator ^ (const L & l, const R & r)

Exclusive disjunction is a binary relation between any two relations with a meaning similar to “either or” in English. It is denoted by operator ^ in Castor. Exclusive disjunction is itself a relation which takes two other relations as arguments. In other words, it is a higher order binary relation. A simplified definition for exclusive disjunction is: a relation that succeeds when one of its argument relations succeeds. The second argument is evaluated only if the first does not succeed. To accommodate the ability of its argument relations to evaluate successfully zero or more times, we broaden its definition to: a relation that succeeds each time one of its argument relations succeeds. Disjunction can itself succeed zero, one or more times depending upon its argument relations. For instance consider the following simple expression:

range(x, 1, 3) ^ range(x, 7, 10)

Here both arguments to ^ are range relations. The first relation indicates that x is a value in the inclusive range 1 through 3. The second relation indicates that x is a value in the inclusive range 7 through 10. The intent here is to state that x can have a value that is either between 1 and 3 or between 7 and 10, and not in both ranges. If x is left undefined, only the first range relation will produce values for x. Due to successful evaluation of first relation, the second range relation will be ignored. However if x’s value is defined such that it’s value in the range 7 through 10, the first relation will fail leading to the evaluation of the second range relation.

Operational semantics of operator ^ can be summarized as: for every successful evaluation of l make sure r does not succeed and vice versa. The following pseudo code demonstrates this.

//This is psuedo code!

bool lhsSucceded = false;

while( lhs() ) {

lhsSucceded = true;

***yield*** return true; // Not C++. ‘yield’ keyword is not valid C++

}

while(!lhsSucceded && rhs())

***yield*** return true;

return false;

In terms of the imperative paradigm, the operational semantics can be visualized as two main loops, one following another as shown above. The first loop is responsible for find a successful evaluation of lhs and the second loop for rhs. A boolean flag is used to ensure that rhs is attempted only if lhs never succeeded. This loop continues to succeed as long lhs or rhs succeeds.

## Recursion

**recurse relation**

// support for nullary relations

template<typename Rel\_0>

Recurse0\_r<Rel\_0>

**recurse**( Rel\_0 r )

// remaining overloads support relations with up to 6 arguments

template<typename Rel\_1, typename A1>

Recurse1\_r<Rel\_1,A1>

**recurse**(Rel\_1 r, lref<A1>& a1)

template<typename Rel\_2, typename A1, typename A2>

Recurse2\_r<Rel\_2,A1,A2>

**recurse**(Rel\_2 r, lref<A1>& a1, lref<A2>& a2)

template<typename Rel\_3, typename A1, typename A2, typename A3>

Recurse3\_r<Rel\_3,A1,A2,A3>

**recurse**(Rel\_3 r, lref<A1>& a1, lref<A2>& a2, lref<A3>& a3)

template<typename Rel\_4, typename A1, typename A2, typename A3

, typename A4>

Recurse4\_r<Rel\_4,A1,A2,A3,A4>

**recurse**(Rel\_4 r, lref<A1>& a1, lref<A2>& a2, lref<A3>& a3

, lref<A4>& a4)

template<typename Rel\_5, typename A1, typename A2, typename A3

, typename A4, typename A5>

Recurse5\_r<Rel\_5,A1,A2,A3,A4,A5>

**recurse**(Rel\_5 r, lref<A1>& a1, lref<A2>& a2, lref<A3>& a3

, lref<A4>& a4, lref<A5>& a5)

template<typename Rel\_6, typename A1, typename A2, typename A3

, typename A4, typename A5, typename A6>

Recurse6\_r<Rel\_6,A1,A2,A3,A4,A5,A6>

**recurse**(Rel\_6 r, lref<A1>& a1, lref<A2>& a2, lref<A3>& a3

, lref<A4>& a4, lref<A5>& a5, lref<A6>& a6)

**Declarative reading**: Same as r(a1,..,aN).

**Template Parameters:**

Rel\_*N* : A function pointer to relation that takes *N* arguments. All parameter types must be logic references.

A*n* : Effective type of the *N*th argument to being passed. Can be a POT or lref whose effective type is convertible to the corresponding parameter type in Rel\_*N*.

**Parameters:**

r : Relation (with up to 6 parameters) to be recursed on. This should be the same as the relation or instance of function object within which the call to recurse is made.

a*N\_* : [in/out] The Nth argument to r. Must be a logic reference.

**Exceptions:**

Same as those thrown by relation r.

**Notes:** Relation recurse provides the mechanism for making recursive calls inside relations. Consider the following well intentioned, but erroneous, recursive call within ancestor relation.

// ans is ancestor of descendant des

relation **ancestor**(lref<string> ans, lref<string> des) {

lref<string> X;

return parent(ans,des)

|| parent(X,des) && ancestor(ans,X);

}

The problem with the above definition is that the below attempt to use ancestor will lead to infinite recursion:

relation r = ancestor("Leda","Castor"); // will never return!

r(); // execution will not reach here

The recursion in this case should only be performed when the relation is actually evaluated by executing r(). This problem can be resolved by using relation recurse to delay the recursive call in ancestor as follows:

relation **ancestor**(lref<string> ans, lref<string> des) {

lref<string> X;

return parent(ans,des)

|| parent(X,des) && **recurse**(&ancestor,ans,X);

}

**Also refer to:**

recurse\_mf.

**recurse\_mf relation**

// support for nullary member relations

template<typename Obj, typename MemRel\_0>

RecurseMem0\_r<Obj,MemRel\_0>

**recurse\_mf**(Obj\* obj, MemRel\_0 mr)

// remaining overloads support member relations with up to 6 arguments

template<typename Obj, typename MemRel\_1, typename A1>

RecurseMem1\_r<Obj,MemRel\_1,A1>

**recurse\_mf**(Obj\* obj, MemRel\_1 mr, lref<A1>& a1)

template<typename Obj, typename MemRel\_2, typename A1, typename A2>

RecurseMem2\_r<Obj,MemRel\_2,A1,A2>

**recurse\_mf**(Obj\* obj, MemRel\_2 mr, lref<A1>& a1, lref<A2>& a2)

template<typename Obj, typename MemRel\_3, typename A1, typename A2, typename A3>

RecurseMem3\_r<Obj,MemRel\_3,A1,A2,A3>

**recurse\_mf**(Obj\* obj, MemRel\_3 mr, lref<A1>& a1, lref<A2>& a2

, lref<A3>& a3)

template<typename Obj, typename MemRel\_4, typename A1, typename A2, typename A3, typename A4>

RecurseMem4\_r<Obj,MemRel\_4,A1,A2,A3,A4>

**recurse\_mf**(Obj\* obj, MemRel\_4 mr, lref<A1>& a1, lref<A2>& a2

, lref<A3>& a3, lref<A4>& a4)

template<typename Obj, typename MemRel\_5, typename A1, typename A2, typename A3, typename A4, typename A5>

RecurseMem5\_r<Obj,MemRel\_5,A1,A2,A3,A4,A5>

**recurse\_mf**(Obj\* obj, MemRel\_5 mr, lref<A1>& a1, lref<A2>& a2

, lref<A3>& a3, lref<A4>& a4, lref<A5>& a5)

template<typename Obj, typename MemRel\_6, typename A1, typename A2, typename A3, typename A4, typename A5, typename A6>

RecurseMem6\_r<Obj,MemRel\_6,A1,A2,A3,A4,A5,A6>

**recurse\_mf**(Obj\* obj, MemRel\_6 mr, lref<A1>& a1, lref<A2>& a2

, lref<A3>& a3, lref<A4>& a4, lref<A5>& a5

, lref<A6>& a6)

**Declarative reading**: Same as (obj->\*mr)(a1,..,aN).

**Template Parameters:**

Obj : Type whose member relation is to be recurse.

MemRel\_*N* : Pointer to member relation in Obj that takes up to *N* arguments. All parameter types must be logic references.

A*n* : Effective type of the Nth argument to being passed. If lref<P*n>* is the type of the *N*th parameter of relation MemRel\_*N*, then A*n* should be same as P*n*.

**Parameters:**

obj : Object whose method member relation identified by mr will be recursed on. This argument should always be the this.

mr : Member relation (with up to 6 parameters) to be recursed on. This should be the same as the member relation within which the call to recurse\_mf is made.

a*N* : [in/out] The Nth argument to mr. Must be a logic reference.

**Exceptions:**

Same as those thrown by mr.

**Notes:** This relation is similar to recurse\_f, except that it used for recursing inside member relations.

**Also refer to:**

recurse\_f.

## Dynamic relations

**Introduction**

Types Conjunctions, Disjunctions and ExDisjunctions together provide a facility to define relations dynamically at runtime. These types are themselves relations and thus can be mixed in with any other relations either defined statically or dynamically. Any relation that can be implemented statically can also be implemented dynamically. Ability to define relations at runtime allows us to define relations based on data obtained at runtime from, for instance, a SQL query or a text file. This also naturally provides a basis for runtime metaprogramming in the Logic paradigm.

**Conjunctions relation**

class Conjunctions {

public:

Conjunctions();

template<typename Rel> void push\_back(const Rel& r);

template<typename Rel> void push\_front(const Rel& r);

bool operator ()(void);

};

**Brief Description**: Represents relational conjunction expression to which clauses can be added at runtime.

**Methods**

**Conjunctions**()

Constructs a Conjunctions relation with no clauses. An empty Conjunctions relation will fail on evaluation.

template<typename Rel> void **push\_back**(const Rel& r)

Adds the clause r at the back. Rel is any type that can be treated as a relation.

template<typename Rel> void **push\_front**(const Rel& r)

Adds the clause r at the front. Rel is any type that can be treated as a relation.

bool **operator()**(void)

Triggers the evaluation of clauses added to Conjunctions. The contained relations are treated as if an operator && exists between each adjacent relation.

**Examples**

In the following example, all three relations are semantically identical.

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

return eq(p,"Runa") && eq(g,"female");

}

Conjunctions **genderDynamic1**(lref<string> p, lref<string> g) {

Conjunctions result;

result.push\_back( eq(p,"Runa") );

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

return result;

}

Conjunctions **genderDynamic2**(lref<string> p, lref<string> g) {

Conjunctions result;

result.push\_front( eq(g,"female") );

result.push\_front( eq(p,"Runa") );

return result;

}

**Also refer to**

Disjunctions, ExDisjunctions.

**Disjunctions relation**

class Disjunctions {

public:

Disjunctions();

template <typename Rel> void push\_back(const Rel& r);

template <typename Rel> void push\_front(const Rel& r);

bool operator ()(void);

};

**Brief Description**: Represents relational inclusive disjunction expression to which clauses can be added at runtime.

**Methods**

**Disjunctions**()

Constructs a Disjunctions relation with no clauses. An empty Disjunctions relation will fail on evaluation.

template<typename Rel> void **push\_back**(const Rel& r)

Adds the clause r at the back. Rel is any type that can be treated as a relation.

template<typename Rel> void **push\_front**(const Rel& r)

Adds the clause r at the front. Rel is any type that can be treated as a relation.

bool **operator()**(void)

Triggers the evaluation of clauses added to Disjunctions. The contained relations are treated as if an operator || exists between each adjacent relation.

**Examples**

In the following example, all three relations are semantically identical.

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

return eq(p, "Roshan") && eq(g,"male")

|| eq(p, "Runa") && eq(g,"female");

}

Disjunctions **genderDynamic1**(lref<string> p, lref<string> g) {

Disjunctions result;

result.push\_back( eq(p,"Roshan") && eq(g,"male") );

result.push\_back( eq(p,"Runa") && eq(g,"female") );

return result;

}

Disjunctions **genderDynamic2**(lref<string> p, lref<string> g) {

Disjunctions result;

result.push\_front( eq(p,"Runa") && eq(g,"female") );

result.push\_front( eq(p,"Roshan") && eq(g,"male") );

return result;

}

**Also refer to**

Conjunctions, ExDisjunctions.

**ExDisjunctions relation**

class ExDisjunctions {

public:

ExDisjunctions();

template <typename Rel> void push\_back(const Rel& r);

template <typename Rel> void push\_front(const Rel& r);

bool operator ()(void);

};

**Brief Description**: Represents relational exclusive disjunction expression to which clauses can be added at runtime.

**Methods**

**ExDisjunctions**()

Constructs an ExDisjunctions relation with no clauses. An empty ExDisjunctions relation will fail on evaluation.

template<typename Rel> void **push\_back**(const Rel& r)

Adds the clause r at the back. Rel is any type that can be treated as a relation.

template<typename Rel> void **push\_front**(const Rel& r)

Adds the clause r at the front. Rel is any type that can be treated as a relation.

bool **operator()**(void)

Triggers the evaluation of clauses added to ExDisjunctions. The contained relations are treated as if an operator ^ exists between each adjacent relation.

**Examples**

In the following example, all three relations are semantically identical.

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

return **(** eq(p, "Roshan") && eq(g,"male") **)**

**^** **(** eq(p, "Runa") && eq(g,"female") **)**;

}

ExDisjunctions **genderDynamic1**(lref<string> p, lref<string> g) {

ExDisjunctions result;

result.push\_back( eq(p,"Roshan") && eq(g,"male") );

result.push\_back( eq(p,"Runa") && eq(g,"female") );

return result;

}

ExDisjunctions **genderDynamic2**(lref<string> p, lref<string> g) {

ExDisjunctions result;

result.push\_front( eq(p,"Runa") && eq(g,"female") );

result.push\_front( eq(p,"Roshan") && eq(g,"male") );

return result;

}

**Also refer to**

Conjunctions, Disjunctions.

# Inline Logic Reference Expressions (ILE)

ILEs are expressions composed of one or more logic references and most of the common overloadable operators. In C++, ordinarily, an expression returns the result immediately on evaluation. Evaluation of an ILE yields a function object (more precisely, an expression tree) that represents the semantics of the expression. Such function objects can be evaluated at a later point in time by applying the function call operator without arguments. The typical use of ILEs is to declaratively create simple anonymous functions for use as arguments to relations such as eq\_f, write\_f, predicate etc. Consider printing all numbers in an array that match some criteria:

bool lessThan5(lref<int> i) {

return \*i<5;

}

int nums[] = {8,2,9,4,0};

lref<int> i;

relation r = item(i, nums+0, nums+5) && predicate(lessThan5, i);

while(r())

cout << \*i << " ";

The predicate function lessThan5 can be substituted with an ILE declared directly inline with the call to predicate:

relation r = item(i, nums+0, nums+5) && predicate(**i<5**);

while(r())

cout << \*i << " ";

Just like functions, ILEs can be classified as pure or impure. ILEs that induce side effects on any externally visible objects or logic references are impure. Typically such ILEs consist of operators such as ++. Impure ILEs, functions or member functions should not be used to create relations as they interfere with backtracking which relies on restoration of any state change. As side effects induced by ILE arguments to relations are not undone automatically during backtracking, it can lead to improper evaluation of relations.

## Operator overloads for creating ILEs:

The listing below describes the overloads defined for binary operator + for creating ILEs. Similar overloads are also provided for the binary operators +, -, \*, /, %, |, ^, &, <<, >>, ==, != , <, >, <=, >=, && and ||.

template <typename L, typename R>

Ile<Plus\_ILE<lref<L>, R> >

operator **+** (lref<L> & left, const R & right);

template <typename L, typename R>

Ile<Plus\_ILE<L, lref<R> > >

operator **+** (const L & left, lref<R> & right);

template <typename L, typename R>

Ile<Plus\_ILE<lref<L>, lref<R> > >

operator **+** (lref<L> & left, lref<R> & right);

template <typename L, typename R>

Ile<Plus\_ILE<Ile<L>, R> >

operator **+** (const Ile<L> & left, const R & right);

template <typename L, typename R>

Ile<Plus\_ILE<L, Ile<R> > >

operator **+** (const L & left, const Ile<R> & right);

template <typename L, typename R>

Ile<Plus\_ILE<Ile<L>, Ile<R> > >

operator **+** (const Ile<L> & left, const Ile<R> & right);

template <typename L, typename R>

Ile<Plus\_ILE<Ile<L>, lref<R> > >

operator **+** (const Ile<L> & left, lref<R> & right);

template <typename L, typename R>

Ile<Plus\_ILE<lref<L>, Ile<R> > >

operator **+** (lref<L> & left, const Ile<R> & right);

Template type Plus\_ILE in the above listing represents an internal type used to represent a node in the expression tree corresponding to operator +. It implements operator() evaluating the particular node it represents in the expression tree.

The listing below describes the overloads defined for the prefix unary operator + for creating ILEs. Overloads similar to the following are also provided for prefix unary operators -, ~, !, ++, and --.

template <typename T>

Ile<Prefix\_Plus\_ILE<lref<T> > >

operator **+** (lref<T> const & obj);

template <typename T>

Ile<Prefix\_Plus\_ILE<Ile<T> > >

operator **+** (Ile<T> const & expr);

The listing below describes the overloads defined for the postfix unary operator ++ for creating ILEs. Overloads similar to the following are also provided for postfix unary operator --.

template <typename T>

Ile<Postfix\_Inc\_ILE<lref<T> > >

operator **++** (lref<T> const & obj, int);

template <typename T>

Ile<Postfix\_Inc\_ILE<Ile<T> > >

operator **++** (Ile<T> const & expr, int);

Note that all overloadable operators with the exception of the following are supported for the creation of ILEs from lrefs.

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

## Named ILEs

Since there are only a limited number of operators in C++, additional ILE operators can be introduced in the form of named functions instead. The named ILEs can be freely combined with other ILE operators in an ILE expression.

**call**

// For nullary through sestiary(6-ary) functions

template<typename FuncT>

Ile<Call\_0<...> >

**call**(FuncT& f)

template<typename FuncT, typename A1>

Ile<Call\_1<...> >

**call**(FuncT& f, const A1& a1)

template<typename FuncT, typename A1, typename A2>

Ile<Call\_2<...> >

**call**(FuncT& f, const A1& a1, const A2& a2)

.. additional overloads supporting upto 6 arguments to f

// For nullary through sestiary(6-ary) function objects

template<typename Ret>

Ile<Call\_0<...> >

**call**(Ret(\* f)(void))

template<typename Ret, typename P1, typename A1>

Ile<Call\_1<...> >

**call**(Ret(\* f)(P1), const A1& a1)

template<typename Ret, typename P1, typename P2

, typename A1, typename A2>

Ile<Call\_2<...> >

**call**(Ret(\* f)(P1,P2), const A1& a1, const A2& a2)

.. additional overloads supporting upto 6 arguments to f

**Brief Description**: call is used to create an ILE function object that on evaluation invokes the function or function object f.

**Template Parameters:**

FuncT: Type of the function object to be invoked. FuncT should be copy constructible.

Ret: Return type function to be invoked.

A1..A*n* : Types of the arguments to be passed to the function or function object. Can be a lref or a POT.

P1..P*n* : Parameters types of the function. Any A*n* should be either same as or convertible to the corresponding P*n*. Can be a lref or a POT.

**Parameters:**

f: The function or function object to invoked. Note that if f is a function object, the actual invocation will occur on a copy of f.

a1..a*N* : Arguments to be passed to f. Arguments can be lrefs or POTs. Lref arguments (if any) will be automatically dereferenced during invocation of f.

**Returns:**

A function object, which on evaluation returns the value produced by invoking f.

**Notes**:

call supports invocation of functions and function objects with up to 6 arguments. The function object returned by call contains a *copy* of all its arguments (including f) to call. Arguments are always passed to f *by- value*. This is the case regardless of whether the arguments to call or the f’s parameter types are regular C++ references or logic references.

**Exceptions:**

Any exception thrown by invocation of f.

Any exception thrown during copy construction of some A*N*, OR, during the conversion of A*N* to P*N* (if A*N* is not same as P*N*).

Any exception thrown by copy construction of f if f is a function object.

**Examples**:

int squareroot(int i) {

return ...;

}

int arr[] = {4,9,16,25,36};

// Basic standalone usage – calling squareroot()

lref<int> sr,i;

relation r = item(i,arr,arr+5) && eq\_f(sr, call(squareroot,i) );

while(r())

cout << \*sr << " ";

// Compound expressions – computing square of the square root

lref<int> j,s;

relation r2 = item(j,arr,arr+5)

&& eq\_f(s, call(squareroot,j)\*call(squareroot,j) );

while(r2())

cout << \*s << " ";

**create**

template<typename T>

Ile<CreateWith0<T> >

**create**()

template<typename T, typename A1>

Ile<CreateWith1<T,A1> >

**create**(const A1& a1)

template<typename T, typename A1, typename A2>

Ile<CreateWith2<T,A1,A2> >

**create**(const A1& a1, const A2& a2)

.. additional overloads supporting upto 6 arguments

**Brief Description**: create is used to create an ILE function object that on evaluation returns an object of type T. Arguments to create are forwarded to T’s constructor.

**Template Parameters:**

T: Type of object to be constructed. Depending upon the specific overload of create used, T must support a constructor with accepts arguments of types A1 … AN. This type must be explicitly specified.

A1..A*N* : Types of the arguments to be passed to T’s constructor when creating object of type T. Since these types are automatically inferred by the compiler there is no need to specify them explicitly.

**Parameters:**

a1..a*N* : Arguments to be passed to T’s constructor when creating object of type T.

**Returns:**

A function object, which on evaluation returns an object of type T.

**Notes**:

create supports construction of objects with up to 6 arguments. It can be combined with other ILE operators or named ILEs to create more complex ILEs/function objects. For e.g. create<complex<int> >(1,4) \* 2 creates an ILE or function object that multiples the complex number (1,4) with 2.

**Exceptions:**

Any exception thrown by T’s constructor.

**Examples**:

// if i is item in arr1, and j is item in arr2,

// generate std::pair<int,int> in p such that i+j==4

int arr1[] = { 1 , 0 , 5, 3 };

int arr2[] = { 2 , 4 ,-1, 5 };

lref<pair<int,int> > p;

lref<int> i, j;

relation r = item(i, arr1+0, arr1+4) && item(j, arr2+0, arr2+4)

&& predicate(i+j==4)

&& eq\_f( p, **create**<pair<int,int> > (i,j) );

**Create::with [deprecated, use create()]**

template<typename T>

class Create {

public:

static Ile<CreateWith0<T> >

**with**();

template<typename A1>

static Ile<CreateWith1<T,A1> >

**with**(const A1& a1);

template<typename A1, typename A2>

static Ile<CreateWith2<T,A1,A2> >

**with**(const A1& a1, const A2& a2);

.. additional overloads supporting upto 6 arguments to f

};

**Brief Description**: Create<T>::with is used to create an ILE function object that on evaluation returns an object of type T. Arguments to with() are forwarded to T’s constructor.

**Template Parameters:**

T: Type of object to be constructed. Depending upon the overload of which that is used, T must support a constructor that accepts arguments of types A1 … AN.

A1..A*N* : Types of the arguments to be passed to T’s constructor when creating object of type T.

**Parameters:**

a1..a*N* : Arguments to be passed to T’s constructor when creating object of type T.

**Returns:**

A function object that on evaluation returns an object of type T.

**Notes**:

Create::with supports construction of objects with up to 6 arguments. It can be combined with other ILE operators or named ILEs to create more complex ILEs/function objects. For e.g. Create<complex<int> >::with(1,4) \* 2 creates an ILE or function object that multiples the complex number (1,4) with 2.

**Exceptions:**

Any exception thrown by T’s constructor

**Examples**:

// if i is item in arr1, and j is item in arr2,

// generate std::pair<int,int> in p such that i+j==4

int arr1[] = { 1 , 0 , 5, 3 };

int arr2[] = { 2 , 4 ,-1, 5 };

lref<pair<int,int> > p;

lref<int> i, j;

relation r = item(i, arr1+0, arr1+4) && item(j, arr2+0, arr2+4)

&& predicate(i+j==4)

&& eq\_f( p, **Create**<pair<int,int> >::**with**(i,j) );

**get**

template<typename Obj, typename MemberT>

Ile<Get<Obj,MemberT> >

**get**(const lref<Obj>& obj\_, MemberT Obj::\* mem)

**Brief Description**: get is used to create an ILE function object that on evaluation returns a reference to a data member of obj\_.

**Template Parameters:**

Obj : Any type which satisfies requirements of logic reference.

MemberT : The actual type of the data member to be accessed in class Obj. The type of the pointer to data member is MemberT Obj:: \*

**Parameters:**

obj\_ : The object whose data member is to be accessed. This argument must be a logic reference. This restriction ensures methods are invoked on the actual argument and not on a copy of obj\_.

mem : Pointer to a data member of obj\_.

**Returns:**

A reference to (\*obj\_).\*mem

**Notes**:

It can be combined with other ILE operators or named ILEs to create more complex ILEs/function objects as shown in the example below.

**Exceptions:**

InvalidDeref : If obj\_ is not initialized at the time of evaluation.

**Examples**:

struct Name {

string firstName, lastName;

Name(string firstName, string lastName)

: firstName(firstName), lastName(lastName)

{ }

bool operator==(const Name& rhs) {

return firstName==rhs.firstName && lastName==rhs.lastName;

}

};

// print all first names from a list<Name>

lref<Name> n;

lref<list<Name> > names = ...;

relation printFname = item(n, names)

&& write\_f( get(n, &Name::firstName) + string(" ") );

while(printFname());

**mcall**

// For nullary through sestiary(6-ary) member functions

template<typename R, typename Obj>

Ile<MCall\_r0<Obj,R(Obj::\*)(void),R> >

**mcall**(lref<Obj>& obj\_, R(Obj::\*mf)(void) )

template<typename R, typename P1, typename Obj, typename A1>

Ile<MCall\_r1<Obj,R(Obj::\*)(P1),R,A1> >

**mcall**(lref<Obj>& obj\_, R(Obj::\* mf)(P1), const A1& a1\_)

template<typename R, typename P1, typename P2, typename Obj

, typename A1, typename A2>

Ile<MCall\_r2<Obj,R(Obj::\*)(P1,P2),R,A1,A2> >

**mcall**(lref<Obj>& obj\_, R(Obj::\* mf)(P1,P2), const A1& a1\_

, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to mf

// For nullary through sestiary(6-ary) const member functions

template<typename R, typename Obj>

Ile<MCall\_r0<Obj,R(Obj::\*)(void) const,R> >

**mcall**(lref<Obj>& obj\_, R(Obj::\*mf)(void) const)

template<typename R, typename P1, typename Obj, typename A1>

Ile<MCall\_r1<Obj,R(Obj::\*)(P1) const,R,A1> >

**mcall**(lref<Obj>& obj\_, R(Obj::\* mf)(P1) const, const A1& a1\_)

template<typename R, typename P1, typename P2, typename Obj

, typename A1, typename A2>

Ile<MCall\_r2<Obj,R(Obj::\*)(P1,P2) const,R,A1,A2> >

**mcall**(lref<Obj>& obj\_, R(Obj::\* mf)(P1,P2) const, const A1& a1\_

, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to mf

**Brief Description**: mcall is used to create an ILE function object that on evaluation invokes the member function mf on obj\_.

**Template Parameters:**

R: Return type of member function to be invoked.

P1..P*n* : Parameters types of the member function. Can be a lref or a POT.

Any A*n* should be either same as or convertible to the corresponding P*n*.

A1..A*n* : Types of the arguments to be passed to the member function. Can be a lref or a POT.

**Parameters:**

mf: Pointer to a member function.

a1..a*N* : Arguments to be passed to mf. Arguments can be lrefs or POTs. Lref arguments (if any) will be automatically dereferenced during invocation of f.

**Returns:**

A function object, which on evaluation returns the value produced by invoking mf.

**Notes**:

mcall supports invocation of member functions with up to 6 arguments. The function object returned by mcall contains a copy of all its arguments. Arguments are always passed to mf *by- value*. This is the case regardless of whether the arguments to mcall or the mf’s parameter types are regular C++ references or logic references.

**Exceptions:**

Any exception thrown by invocation of mf.

Any exception thrown during copy construction of some A*N*, OR, during the conversion of A*N* to P*N* (if A*N* is not same as P*N*).

**Examples**:

// print non empty strings

vector<string> values = ... ;

lref<string> s;

relation r = item(s,values.begin(),values.end())

&& predicate( mcall(s,&string::length)>0 );

while(r())

cout << \*s << "\n";

**ref**

template<typename T>

Ile<Ref<T> > ref(T& obj)

**Brief Description**: ref is used to create an ILE function object that on evaluation returns a reference to obj.

**Template Parameters:**

T: Any type.

**Parameters:**

obj: The object to which a reference is to be taken.

**Returns:**

Reference to obj.

**Exceptions:**

None.

**Notes**:

The closure of an ILE is the collection of all objects (lrefs and regular variables) referenced in the ILE. All objects referenced in an ILE are stored *by value* in the function object representing the ILE. In other words, by default, the closure of an ILE has lvalue semantics. However, since copy construction of an lref creates a coreference, all lrefs in the closure effectively exhibit lvalue semantics. Therefore we can say that all lrefs in the closure exhibit lvalue semantics and all other objects exhibit rvalue semantics. ILE ref is useful in changing the rvalue semantics of regular objects to lvalue semantics. This allows us to use objects that do not allow copy construction (e.g. std::cout) in an ILE. Users must ensure that obj continues to exist at the time when the ILE undergoes evaluation; otherwise it results in undefined behavior.

**Examples**:

//create an ILE that prints to cout values of x

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

lref<int> x;

relation r = item(x,a,a+3) && eval( ref(cout)<< x << string(" ") );

while(r());

## Return types for ILE operators

Since there is no way, currently, to programmatically deduce return types of arbitrary functions or operators in C++, the following assumptions are made about the return types of overloaded operators used in creating ILEs.

* All comparison operators (<, >=, ==, != etc.) and logical operators &&, || and ! have return type bool.
* Return type of prefix operators ++ 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.

If operators defined on certain types do not conform to the above assumptions and they need to be use in an ILE expression, wrap the sub expression involving such operators into a regular function and invoke it via call.

# Higher-Order relations

**zip relation**

template<typename L, typename R>

Zip\_r<L,R> **zip**(const L& l, const R& r)

**Declarative reading**: Succeed until both relations l and r succeed.

**Template Parameters:**

L, R: Any type that can be treated as a relation.

**Parameters:**

l: The first relation to be evaluated.

r: The second relation to be evaluated.

**Exceptions:**

Any exception thrown by operator >>.

**Notes**:

Relation zip provides a facility for interleaved evaluation of relations l and r. Each evaluation zip will evaluate relations l and r once, in that order. Relation r will be evaluated only if l succeeds. If both l and r succeed then zip succeeds else zip will fail.

**Examples**:

// print parallel sum of two arrays

lref<int> i, j, sum;

int ai[] = { 1,2,3,4 }, aj[] = { 1,2,3,4,5,6,7,8 };

relation r = **zip(** item(i,ai,ai+4), item(j,aj,aj+5) **)** && eq\_f(sum,i+j);

while( r() )

cout << \*sum << " "; // prints: 2,4,6,8

# Utils

## Input/Output relations

**read relation**

template <typename T>

Read\_r<T> **read** (lref<T> val)

Read\_r<std::string> **read**(const char\* val)

**Declarative reading**: val is the value read from std::cin.

**Template Parameters:**

T : A type that supports reading from std::cin using operator <<.

**Parameters:**

val : [in/out] The value to be read. If not initialized, it will be assigned the value that is read. If initialized, it is compared with the value read from stream.

**Exceptions:**

Any exception thrown by operator >>.

**Notes**:

Relation read provides a relational facility for reading from std::cin. The action of reading values from std::cin will not be reverted during backtracking. Relations read\_f and read\_mf (similar to the write\_f and write\_mf counterparts) are not provided since eq\_f and eq\_mf already provide this functionality.

**Examples**:

// 1) read words from std::cin and echo them to std::cout

lref<string> str;

relation r = read(str);

while(r**()**)

cout << \*str << "\n";

// 2) read a word from input and check if it is "Logic"

if(read("Logic")**()**){

...

}

**Also refer to:**

readFrom, write, writeTo, eq, eq\_f, eq\_mf

**readFrom relation**

template <typename T>

Read\_r<T> **readFrom**(std::istream& inputStream, lref<T> val)

Read\_r<std::string> **readFrom**( std::istream& inputStream

, const char\* val)

**Declarative reading**: val is the value read from inputStream.

**Template Parameters:**

T : A type that supports reading from an inputStream (like std::cin) using operator <<.

**Parameters:**

inputStream : The stream from which a value is to be read. Defaults to std::cin.

val : [in/out] The value to be read. If not initialized, it will be assigned the value that is read. If initialized, it is compared with the value read from stream.

**Exceptions:**

Any exception thrown by operator >>.

**Notes**:

This relation is similar to relation read, but the is parameterized on the input stream from which value is to be read.

**Examples**:

// 1) read from strstream and std::cin until words from both match

strstream sstrm;

sstrm << "Words in this sentence are expected";

lref<string> str;

relation r = readFrom(sstrm,str) && read(str);

while(r());

// 2) Copy all words from a strstream to cout

strstream sstrm;

sstrm << "Writing must be learnt by wrote";

lref<string> str;

relation r = readFrom(str,sstrm) && write(str) && write(" ");

while(r());

**Also refer to:**

read, write, writeTo, eq, eq\_f, eq\_mf

**write relation**

template <typename T>

Write\_r<T> **write**(const T& obj\_)

Write\_r<std::string> **write**(const char\* obj\_)

**Declarative reading**: Write obj to std::cout.

**Template Parameters:**

T : Can be a logic reference, or any other POT. The effective type should support writing to std::cout using operator <<. T must support copy construction.

**Parameters:**

obj\_ : [in] The object to be printed. Can be a lref or POT

**Exceptions:**

InvalidDeref : If obj\_ is an uninitialized logic reference at the time of evaluation.

Any exception thrown by operator <<.

**Notes**:

Relation write provides a simple relational facility for writing to std::cout. The action of printing values will not be reverted during backtracking. write evaluates successfully only once.

The second overload provides special case handling for char\* arguments by treating them as strings instead of pointer to a character. This enables usage of write relation in context of char\* arguments more directly as write(“hello”) instead of write<string>(“hello”).

**Examples**:

// 1) printing strings or other types

write("Hello world.")**()**; // prints “Hello world.”

relation r= write("Hello ") && write("world.");

r(); // prints “Hello world.”

write(2.5)**()**; // prints “2.5”

// 2) printing values of logic references.

lref<int> li = 4;

write(li)**()**; // prints “4, ”

**Also refer to:**

write\_f, write\_mf, writeAll, read

**writeTo relation**

template <typename T>

Write\_r<T> **writeTo**(std::ostream& outputStrm, T& obj\_)

Write\_r<std::string> **writeTo**(std::ostream& outputStrm, const char\* obj\_)

**Declarative reading**: write obj to outputStrm.

**Template Parameters:**

T : Can be a logic reference, or any POT that supports writing to a ostream using operator <<. T’s effective type should support writing to ostream using operator <<. T must support copy construction.

**Parameters:**

outputStrm: The output stream to which obj\_ will be printed.

obj\_ : [in] The object to be printed.

**Exceptions:**

InvalidDeref : If obj\_ is a lref that is not initialized at the time of evaluation.

Any exception thrown by operator <<.

**Notes**:

Relation write provides a simple relational facility for writing to ostreams. The action of printing values to any ostream will not be reverted during backtracking. write evaluates successfully only once.

The second overload provides special case handling for char\* arguments by treating them as strings instead of pointer to a character. This also enables usage of write relation in context of char\* arguments more directly as write(“hello”) instead of write<string>(“hello”).

**Examples**:

// 1) printing strings or other types

stringstream sstrm;

writeTo(sstrm, "Hello world")**();** // prints “Hello world.”

**write\_f relation**

// overloads for function objects

template<typename Func>

WriteF\_r<Func>

**write\_f**(Func f, std::ostream& outputStrm=std::cout)

template<typename Func1, typename A1>

WriteF1\_r<Func1, A1>

**write\_f**(Func1 f, A1& a1\_)

template<typename Func2, typename A1, typename A2>

WriteF2\_r<Func2, A1, A2>

**write\_f**(Func2 f, const A1& a1\_, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to f

// overloads for function pointers

template<typename R>

WriteF\_r<R(\*)(void)>

**write\_f**(R(\* f)(void))

template<typename R, typename P1, typename A1>

WriteF1\_r<R(\*)(P1),A1>

**write\_f**(R(\* f)(P1), const A1& a1\_)

template<typename R, typename P1, typename P2, typename A1

, typename A2>

WriteF2\_r<R(\*)(P1,P2),A1,A2>

**write\_f**(R(\* f)(P1,P2), const A1& a1\_, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to f

**Declarative reading**: write to std::cout value returned by invoking f (a1\_ .. aN\_).

**Template Parameters:**

Func*N*: A function pointer or function object type with arity *N*. Func*N*’s parameters cannot be logic references.

R: Return type of the function pointer.

P*n*: Type of the Nth parameter of function pointer. Can be an lref or POT. A*N* should be either same as or convertible to the corresponding P*n*.

A*n*: Type of argument passed at position *n* to Func*N*. Can be a POT or lref whose effective type is convertible to the corresponding parameter type in Func*N*.

**Parameters:**

f : The result of evaluating this function object or function pointer will be written to the specified stream.

a*N\_* : [in] Argument (POT or lref) at position *N* whose effective value will be passed to f.

**Exceptions:**

Any exception thrown by operator << when applied to std::cout.

**Notes:**

Although arguments passed to write\_f may be lrefs or POTs.

**Examples:**

// 1) With regular functions

int add(int l, int r) {

return l+r;

}

lref<int> li=2, lj=3;

write\_f(add, li, lj)**();**

// 2) With ILEs

lref<int> li=2, lj=3;

write\_f(li+lj)**();**

**Also refer to:**

write, write\_mf, read

**writeTo\_f relation**

// overloads for function objects

template<typename Func>

WriteF\_r<Func>

**writeTo\_f**(std::ostream& outputStrm,Func f)

template<typename Func1, typename A1>

WriteF1\_r<Func1, A1>

**writeTo\_f**(std::ostream& outputStrm, Func1 f, const A1& a1\_)

template<typename Func2, typename A1, typename A2>

WriteF2\_r<Func2, A1, A2>

**writeTo\_f**(std::ostream& outputStrm, Func2 f, const A1& a1\_

, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to f

// overloads for function pointers

template<typename R>

WriteF\_r<R(\*)(void)>

**writeTo\_f**(std::ostream& outputStrm, R(\* f)(void))

template<typename R, typename P1, typename A1>

WriteF1\_r<R(\*)(P1),A1>

**writeTo\_f**(std::ostream& outputStrm, R(\* f)(P1), const A1& a1\_)

template<typename R, typename P1, typename P2, typename A1

, typename A2>

WriteF2\_r<R(\*)(P1,P2),A1,A2>

**writeTo\_f**(std::ostream& outputStrm, R(\* f)(P1,P2), const A1& a1\_

, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to f

**Declarative reading:** write to outputStrm value returned by invoking f with arguments a1\_ .. aN\_.

**Template Parameters:**

Func*N*: A function pointer/object type with arity *N*. Func*N*’s parameters cannot be logic references.

R: Return type of the function pointer.

P*n*: Type of the Nth parameter of function pointer. Can be an lref or POT. A*N* should be either same as or convertible to the corresponding P*n*.

A*n*: Type of argument passed at position *n* to the Func*N* type. Can be a lref or POT whose effective type is convertible to the corresponding parameter type in Func*N*.

**Parameters:**

f : The result of evaluating this function object or function pointer will be written to the specified stream.

a*N\_* : [in] Argument (POT or lref) at position *N* whose effective value will be passed to f.

outputStrm: Stream to which value will be written.

**Exceptions:**

InvalidDeref : If any aN\_ is a lref that has not been initialized at the time of evaluation.

Any exception thrown by operator <<.

**Notes:**

This relation is similar to write\_f but allows explicit specification of the stream to which data is to be written.

**Also refer to:**

write, write\_f, write\_mf, writeTo\_mf, read, readFrom

**write\_mf relation**

// Overloads for non-const member functions

template<typename R, typename Obj>

WriteMF\_r<Obj,R(Obj::\*)(void)>

**write\_mf**(lref<Obj>& obj\_, R(Obj::\*mf)(void) )

template<typename R, typename P1, typename Obj, typename A1>

WriteMF1\_r<Obj,R(Obj::\*)(P1),A1>

**write\_mf**(lref<Obj>& obj\_, R(Obj::\* mf)(P1), const A1& a1\_)

template<typename R, typename P1, typename P2, typename Obj

, typename A1, typename A2>

WriteMF2\_r<Obj,R(Obj::\*)(P1,P2),A1,A2>

**write\_mf**(lref<Obj>& obj\_, R(Obj::\* mf)(P1,P2), const A1& a1\_

, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to mf

// Overloads for const member functions

template<typename R, typename Obj>

WriteMF\_r<Obj,R(Obj::\*)(void) const>

**write\_mf**(lref<Obj>& obj\_, R(Obj::\*mf)(void) const)

template<typename R, typename P1, typename Obj, typename A1>

WriteMF1\_r<Obj,R(Obj::\*)(P1) const,A1>

**write\_mf**(lref<Obj>& obj\_, R(Obj::\* mf)(P1) const, const A1& a1\_)

template<typename R, typename P1, typename P2, typename Obj

, typename A1, typename A2>

WriteMF2\_r<Obj,R(Obj::\*)(P1,P2) const,A1,A2>

**write\_mf**(lref<Obj>& obj\_, R(Obj::\* mf)(P1,P2) const, const A1& a1\_

, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to mf

**Declarative reading**: write to std::cout the value returned by invoking (\*obj\_).\*mf(a1\_ .. a*N*\_)

**Template Parameters:**

Obj : A type whose member function is to be invoked.

R : Return type of the member function.

P*n*: Type of the nth parameter of member function.

A*n* : Type of the nth argument to being passed. Can be a POT or lref whose effective type is convertible to the corresponding parameter type P*n*.

**Parameters:**

obj\_ : [in] The object whose member function referred to by mf is to be evaluated. This argument must be a logic reference. This restriction ensures methods are invoked on the actual argument and not on a copy of obj\_.

mf : The result of evaluating this member function will be written to the specified stream.

a*N\_* : [in] Argument (POT or lref) at position *N* whose effective value will be passed to mf.

**Exceptions:**

InvalidDeref : If obj\_ or, if any aN\_ is a lref that has not been initialized at the time of evaluation.

Any exception thrown by operator <<.

**Examples:**

// read string from std::cin and print its length

lref<string> s;

relation r = read(s) && write\_mf(s,&string::size);

r();

**Also refer to:**

write, write\_f, writeTo\_f, writeTo\_mf, read, readFrom

**writeTo\_mf relation**

// Overloads for non-const member functions

template<typename R, typename Obj>

WriteMF\_r<Obj,R(Obj::\*)(void)>

**writeTo\_mf**(std::ostream& outputStrm, lref<Obj>& obj\_

, R(Obj::\*mf)(void) )

template<typename R, typename P1, typename Obj, typename A1>

WriteMF1\_r<Obj,R(Obj::\*)(P1),A1>

**writeTo\_mf**(std::ostream& outputStrm, lref<Obj>& obj\_, R(Obj::\* mf)(P1)

, const A1& a1\_)

template<typename R, typename P1, typename P2, typename Obj

, typename A1, typename A2>

WriteMF2\_r<Obj,R(Obj::\*)(P1,P2),A1,A2>

**writeTo\_mf**(std::ostream& outputStrm, lref<Obj>& obj\_

, R(Obj::\* mf)(P1,P2), const A1& a1\_, const A2& a2\_)

// Overloads for const member functions

template<typename R, typename Obj>

WriteMF\_r<Obj,R(Obj::\*)(void) const>

**writeTo\_mf**(std::ostream& outputStrm, lref<Obj>& obj\_, R(Obj::\*mf)(void) const)

template<typename R, typename P1, typename Obj, typename A1>

WriteMF1\_r<Obj,R(Obj::\*)(P1) const,A1>

**writeTo\_mf**(std::ostream& outputStrm, lref<Obj>& obj\_

, R(Obj::\* mf)(P1) const, const A1& a1\_)

template<typename R, typename P1, typename P2, typename Obj

, typename A1, typename A2>

WriteMF2\_r<Obj,R(Obj::\*)(P1,P2) const,A1,A2>

**writeTo\_mf**(std::ostream& outputStrm, lref<Obj>& obj\_

, R(Obj::\* mf)(P1,P2) const, const A1& a1\_, const A2& a2\_)

**Declarative reading**: write to outputStrm the value returned by invoking (\*obj\_).\*mf(a1\_ .. a*N*\_)

**Template Parameters:**

Obj : A type whose member function is to be invoked.

R : Return type of the member function.

P*n*: Type of the nth parameter of member function.

A*n* : Type of the nth argument to being passed. Can be a POT or lref whose effective type is convertible to the corresponding parameter type P*n*.

**Parameters:**

obj\_ : [in] Object on which method is to be invoked. This argument must be a logic reference. This restriction ensures methods are invoked on the actual argument and not on a copy of obj\_.

mf : Pointer to member function whose result will be written to the specified stream.

a*N\_* : [in] Argument (POT or lref) at position *N* whose effective value will be passed to mf.

outputStrm: Stream to which value will be written.

**Exceptions:**

InvalidDeref : If obj\_ or, if any a*N*\_ is a lref that has not been initialized at the time of evaluation.

Any exception thrown by operator <<.

**Notes:**

This relation is similar to write\_mf but allows explicit specification of the stream to which data is to be written.

**Examples:**

// read string from std::cin and print its length to a file

ofstream myfile("example.txt");

lref<string> s = "Castor";

writeTo\_mf(myfile, s, &string::size)**()**;

**Also refer to:**

write, write\_f, write\_mf, writeTo\_f, writeAll, writeAllTo, read, readFrom

**write\_mem relation**

template<typename Obj, typename MemberT>

WriteMem\_r<Obj, MemberT>

**write\_mem**(lref<Obj>& obj\_, MemberT Obj::\* mem)

**Declarative reading**: Write to std::cout the value of (\*obj\_).\*mem

**Template Parameters:**

Obj : Any type which whose member variable is to be accessed.

MemberT : Type of the data member to be accessed.

**Parameters:**

obj\_ : [in] The object whose member variable is to be accessed. This argument must be a logic reference. This restriction ensures methods are invoked on the actual argument and not on a copy of obj\_.

mem : Pointer to a member variable of obj\_ whose value is to be written out to the std::cout.

**Exceptions:**

InvalidDeref : If obj\_ has not been initialized at the time of evaluation.

Also any exception thrown by operator <<.

**Examples:**

typedef pair<string,string> fullname;

lref<fullname> me = fullname("Roshan","Naik");

write\_mem(me, &fullname::first)**()**;

**Also refer to:**

write, write\_f, writeTo\_f, writeTo\_mf, writeTo\_mem, read, readFrom

**writeTo\_mem relation**

template<typename Obj, typename MemberT>

WriteMem\_r<Obj, MemberT>

**writeTo\_mem**(std::ostream& outputStrm, lref<Obj>& obj\_

, MemberT Obj::\* mem)

**Declarative reading**: Write to outputStrm the value of (\*obj\_).\*mem

**Template Parameters:**

Obj : Any type which whose member variable is to be accessed.

MemberT : A Type of the data member to be accessed.

**Parameters:**

obj\_ : [in] The object whose member variable is to be accessed. This argument must be a logic reference. This restriction ensures methods are invoked on the actual argument and not on a copy of obj\_.

mem : Pointer to a member variable of obj\_ whose value is to be written to outputStrm.

outputStrm: Stream to which the value will be written.

**Exceptions:**

InvalidDeref : If obj\_ has not been initialized at the time of evaluation.

Also any exception thrown by operator <<.

**Examples:**

typedef pair<string,string> name;

lref<name> me = name("Roshan","Naik");

stringstream sstrm;

writeTo\_mem(sstrm, me, &name::first)();

**Also refer to:**

write, write\_f, writeTo\_f, writeTo\_mf, writeTo\_mem, read, readFrom

**writeAll relation**

template<typename Itr>

WriteAll\_r<Itr> **writeAll**(Itr begin\_, Itr end\_

, const std::string& separator=" ")

template<typename Cont>

relation **writeAll**(lref<Cont>& cont\_, const std::string& separator=" ")

**Declarative reading**: Write all values in the container cont\_ or in the range [begin\_, end\_) to std::cout.

**Template Parameters:**

Itr : A pointer, iterator, a logic reference to a pointer or a logic reference to a iterator. Dereferencing the effective value of Itr should yield a value that is writable to cout using << operator.

Cont : A container type. Cont should provide methods begin() and end() for obtaining iterators to beginning and one past the end of container. The element type should be writable to cout using << operator.

**Parameters:**

begin\_ : [in] Iterator to the beginning of a sequence to be printed to cout.

end\_ : [in] Iterator to one past the end of a sequence to be printed to cout.

cont\_ : [in] A logic reference to a container whose elements are to be printed to cout.

separator : The string to be printed between two consecutive items in the sequence. Defaults to a single space.

**Exceptions:**

InvalidDeref : If begin\_, end\_ or cont\_ is an uninitialized logic reference at the time of evaluation.

Any exception thrown by operator <<.

**Notes**:

Relation writeAll provides a simple relational facility for writing a sequence of values to std::cout. The action of printing values will not be reverted during backtracking. writeAll evaluates successfully only once.

**Examples**:

// 1) print an array using pointers

string as[] = {"1","2","3","4"};

writeAll(as,as+4)();

// 2) print items in vector (using iterators) using comma as separator

vector<string> vs = ...;

writeAll(vs.begin(),vs.end(), ",");

// 3) print (comma separated) items in vector using lref<iterator>

lref<vector<string> > lvs = vector<string>(as,as+4);

lref<vector<string>::iterator> b,e;

relation r = begin(lvs,b) && end(lvs,e) && writeAll (b,e);

while(r());

// 4) printing values in a container

lref<vector<string> > lvs = vs;

writeAll(lvs)();

**Also refer to:**

write, write\_f, write\_mf, writeTo\_f, writeAllTo, writeTo\_mf, read, readFrom

**writeAllTo relation**

template<typename Itr>

WriteAll\_r<Itr> **writeAllTo**(std::ostream& outputStrm, Itr begin\_

, Itr end\_, const std::string& separator=" ")

template<typename Cont>

relation **writeAllTo**(std::ostream& outputStrm, lref<Cont>& cont\_

, const std::string& separator=" ")

**Declarative reading**: Write all values in the container cont\_ or in the range [begin\_, end\_) to std::cout.

**Template Parameters:**

Itr : A pointer, iterator, a logic reference to a pointer or a logic reference to an iterator. Dereferencing the effective value of Itr should yield a value that is writable to outputStrm using << operator.

Cont : A container type. Cont should provide methods begin() and end() for obtaining iterators to beginning and one past the end of container. The element type should be writable to outputStrm using << operator.

**Parameters:**

outputStrm: Stream to which values will be written.

begin\_ : [in] Iterator to the beginning of a sequence to be printed to cout.

end\_ : [in] Iterator to one past the end of a sequence to be printed to cout.

cont\_ : [in] A logic reference to a container whose elements are to be printed to cout.

separator : The string to be printed between two consecutive items in the sequence. Defaults to a single space.

**Exceptions:**

InvalidDeref : If begin\_, end\_ or cont\_ is an uninitialized logic reference at the time of evaluation.

Any exception thrown by operator <<.

**Notes**:

Relation writeAll provides a simple relational facility for writing a sequence of values to std::cout. The action of printing values will not be reverted during backtracking. writeAll evaluates successfully only once.

**Examples**:

// 1) print an array to stringstream using pointers

string as[] = {"1","2","3","4"};

stringstream sstrm;

writeAllTo(sstrm,as,as+4, " ")();

// 2) print (comma separated) items in vector using lref<iterator>

stringstream sstrm;

lref<vector<string> > lvs = vector<string>(as,as+4);

lref<vector<string>::iterator> b,e;

relation r = begin(lvs,b) && end(lvs,e) && writeAllTo(sstrm,b,e);

while(r());

// 3) printing values in a container

lref<vector<string> > lvs = vs;

writeAll(lvs)();

**Also refer to:**

write, write\_f, write\_mf, writeTo\_f, writeAll, writeTo\_mf, read, readFrom

## Sequences and Containers

**empty relation**

template<typename Cont>

relation **empty**(lref<Cont> c)

template<typename Cont>

relation **empty**(const Cont& c)

**Declarative reading**: Container c is empty.

**Template Parameters:**

Cont : Must satisfy requirements of standard C++ containers [$23.1].

**Parameters:**

c : [in/out] If c is initialized, it will be tested for emptiness, otherwise it will be initialized with an empty container of type Cont.

**Notes:**

Evaluates successfully if container c is empty. If c is initialized, relation empty performs the test for emptiness by comparing c == Cont(). Thus default construction of type Cont is expected to yield an empty container. Similarly if c is not initialized, c will be assigned a default constructed instance of type Cont.

**Also refer to:**

size

**head relation**

template<typename Seq>

Head\_r<Seq> **head**(lref<Seq>& seq\_, lref<typename Seq::value\_type> h)

**Declarative reading**: head of seq\_ is h.

**Template Parameters:**

Seq: Must satisfy the requirements of standard C++ sequences [$23.1.1].

**Parameters:**

seq\_ : [in] Sequence whose head element is of interest.

h : [in/out] The first element in seq\_.

**Exceptions:**

InvalidDeref : If seq\_ is not initialized at the time of evaluation.

**Notes:**

The first element in seq\_ is obtained by dereferencing the begin iterator.

**Also refer to:**

head\_n, tail, tail\_n, head\_tail, head\_n\_tail, next, prev.

**head\_n relation**

template<typename Seq, typename HeadSeq>

relation **head\_n**( lref<Seq>& seq\_

, lref<typename HeadSeq::size\_type> n

, lref<HeadSeq>& h )

**Declarative reading**: h contains first n items from sequence seq\_.

**Template Parameters:**

Seq : Must satisfy requirements of standard C++ sequences [$23.1.1].

HeadSeq : Must satisfy requirements of standard C++ sequences [$23.1.1].

**Parameters:**

seq\_ : [in] Sequence whose first n\_ elements is of interest.

n : [in/out] Number of items in h. 0 <= n <= size of seq\_.

h : [in/out] Sequence containing copies of first n elements from seq\_, i.e. the head sequence.

**Exceptions:**

InvalidDeref : If seq\_ is not initialized at the time of evaluation.

**Notes:**

If n is greater than the number of elements in seq\_, the relation fails. h and seq\_ may be of different types i.e. h can be a list and seq\_ may be a vector.

**Also refer to:**

head, tail, tail\_n, head\_tail, head\_n\_tail.

**head\_tail relation**

template<typename Seq, typename TailSeq>

relation **head\_tail**( lref<Seq>& seq\_

, lref<typename TailSeq::value\_type> h

, lref<TailSeq>& t )

**Declarative reading**: h and t respectively form the head and tail of seq\_.

**Template Parameters:**

Seq : Must satisfy the requirements of standard C++ sequences [$23.1.1].

TailSeq : Must satisfy requirements of standard C++ sequences [$23.1.1].

**Parameters:**

seq\_ : [in] Sequence whose head and tail elements are of interest.

h : [in/out] head of seq\_.

t : [in/out] tail of seq\_.

**Notes:**

h and t are copies of the elements comprising the head and tail of seq\_. t may be a different type than seq\_. head\_tail provides a convenient way to determine head and tail in a single step instead of obtaining them separately using relations head and tail.

**Also refer to:**

head, head\_n, tail, tail\_n, head\_n\_tail.

**head\_n\_tail relation**

template<typename Seq, typename HeadSeq>

relation **head\_n\_tail**( lref<Seq>& seq\_

, lref<typename HeadSeq::size\_type> n

, lref<HeadSeq>& h

, lref<HeadSeq>& t )

**Declarative reading**: h is head sequence and t is tail sequence of seq\_ such that size of h is n.

**Template Parameters:**

Seq : Must satisfy the requirements of standard C++ sequences [$23.1.1].

HeadSeq : Must satisfy requirements of standard C++ sequences [$23.1.1]. This type is the same for both head and tail.

**Parameters:**

seq\_ : [in] Sequence whose head and tail elements are of interest.

n : [in/out] size of head. 0 <= n <= size of seq\_.

h : [in/out] head sequence from seq\_ of size n.

t : [in/out] tail of seq\_.

**Notes:**

h and t are copies of the elements comprising the head and tail of seq\_. Tail t comprises of the all the elements in seq\_ following the head sequence h.

**Also refer to:**

head, head\_n, tail, tail\_n, head\_n\_tail.

**insert relation**

template<typename Seq>

relation **insert**( lref<typename Seq::value\_type> *value\_*

, lref<typename Seq::iterator> *b\_*

, lref<typename Seq::iterator> *e\_*

, lref<Seq>& insertedSeq )

**Declarative reading**: inserting value\_ somewhere into [b\_,e\_) yields sequence insertedSeq.

**Template Parameters:**

Seq: Type representing a sequence of values. Seq must satisfy the requirements of standard C++ sequences [$23.1.1].

**Parameters:**

value\_ : [in] The value to be inserted.

b\_ : [in] Iterator to the start of a sequence of values into which value\_ needs to be inserted.

e\_ : [in] Iterator to one past the end of a sequence of values into which value\_ needs to be inserted.

insertedSeq : [in/out] Sequence containing values from the sequence [b\_,e\_) in addition to value\_. Contains exactly std::distance(b\_,e\_)+1 elements.

**Notes:**

Relative order of values in [b\_,e\_) is preserved in insertedList.

**Examples:**

Number 9 can be inserted into sequence (1,2) in three ways:

(9,1,2), (1,9,2) and (1,2,9). The following code prints each of these combinations.

list<int> li;

li.push\_back(1); li.push\_back(2);

lref<list<int> > insertedSeq;

relation r = insert(9, li.begin(), li.end(), insertedSeq);

while(r()) {

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

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

cout << "\n";

}

**Also refer to:**

insert\_seq, merge.

**insert\_seq relation**

template<typename Seq>

relation **insert\_seq**( lref<typename Seq::iterator> valuesB\_

, lref<typename Seq::iterator> valuesE\_

, lref<typename Seq::iterator> b\_

, lref<typename Seq::iterator> e\_

, lref<Seq>& insertedSeq )

**Declarative reading**: inserting the sequence of values in [valuesB\_,valuesE\_) somewhere into [b\_,e\_) yields sequence insertedSeq.

**Template Parameters:**

Seq: Type representing a sequence of values. Seq must satisfy the requirements of standard C++ sequences [$23.1.1].

**Parameters:**

value\_ : [in] The value to be inserted.

b\_ : [in] Iterator to the start of a sequence of values into which value\_ needs to be inserted.

e\_ : [in] Iterator to one past the end of a sequence of values into which value\_ needs to be inserted.

insertedSeq : [in/out] Sequence containing values from the sequence [b\_,e\_) in addition to value\_. It contains exactly std::distance(b,e)+1 elements.

**Notes:**

Relative order of values in [b\_,e\_) is preserved in insertedSeq. Exact order of values in [valuesB\_,valuesE\_) is preserved in insertedSeq. In other words, inserting the exact sequence [valuesB\_,valuesE\_) at some position into the sequence [b\_,e\_) yields insertedSeq.

**Example:**

Sequence (8,9) can be inserted into sequence (1,2) in three ways:

(8,9,1,2), (1,8,9,2) and (1,2, 8,9). The following code prints each of these combinations.

list<int> li; // sequence to insert into

li.push\_back(1); li.push\_back(2);

list<int> values; // sequence to insert

values.push\_back(8); values.push\_back(9);

lref<list<int> > insertedSeq;

relation r = insert\_seq(values.begin(), values.end()

, li.begin(), li.end(),insertedSeq);

while(r()) {

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

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

cout << "\n";

}

**Also refer to:**

insert, merge.

**merge relation**

template<typename Seq>

relation **merge**(lref<Seq>& l\_, lref<Seq>& r\_, lref<Seq>& m)

**Declarative reading**: Merging sorted sequences l\_ and r\_ yields sorted sequence m.

**Template Parameters:**

Cont : Must satisfy requirements of standard C++ containers [$23.1].

**Parameters:**

l\_, r\_: [in] Sorted sequences to be merged.

m : [in/out] Merged sequence.

**Exceptions:**

InvalidDeref : If l\_ or r\_ are not initialized at the time of evaluation.

**Notes**:

This is the relational equivalent of std::merge.

**Also refer to:**

insert, insert\_seq.

**not\_empty relation**

template<typename Cont>

relation **not\_empty**(lref<Cont> c\_)

template<typename Cont>

relation **not\_empty**(const Cont& c\_)

**Declarative reading**: Container c\_ is not empty.

**Template Parameters:**

Cont : Must satisfy requirements of standard C++ containers [$23.1].

**Parameters:**

c\_ : [in] Container to be tested for emptiness. Must be

**Notes:**

Evaluates successfully if container c\_ is not empty. Unlike relation empty, the container argument c\_ must be initialized at the time of evaluation. Test of emptiness is performed using method Cont::size.

**Also refer to:**

empty, size.

**sequence relation**

template<typename Seq>

Sequence\_r<Seq> **sequence(**lref<Seq>& seq**)**..

**(**lref<T> item**)**..**(**const ConvertibleToT& item**)**..

**(**lref<Seq>& items**)**..

**(**Iter start, Iter end**)..**

**(**LrefIter start, LrefIter end**)**

// where

// T = Seq::value\_type

// LrefIter = lref<Seq::iterator>

// ConvertibleToT = any type convertible to T

**Declarative reading**: seq is a sequence comprising of the arguments following it.

**Template Parameters:**

Seq : Must satisfy requirements of standard C++ sequences [$23.1.1].

**Parameters (Fixed):**

seq : [in/out] The sequence to be unified with the other arguments. It must be a logic reference type. Passing regular container types directly is disabled as it leads to implicitly passing a copy of the original (which can be inefficient) and can also lead to unexpected and surprising behavior.

**Parameters (Variadic):**

After seq, the following parameter types are supported by sequence. Each of these should be separately enclosed in a pair of ( ). Any of the following variadic arguments may be optionally provided and in any order.

a) lref<T> item : Represents a single element in the sequence. This allows passing arguments of type lref<T>.

b) ConvertibleToT& item : Represents a single element in the sequence. This allows passing values of arbitrary types that are convertible to T.

c) lref<Seq>& items : Represents a subsequence of elements occurring in sequence. This allows passing of logic reference to a sequence of the same type as seq. Passing a sequence directly by value is not supported, the argument must be a lref.

d) Iter start, Iter end : A pair of iterators representing a subsequence of elements occurring in seq. This allows passing iterators by value. The iterator type must of type Seq::iterator.

e) LrefIter start, LrefIter end : A pair of logic references to iterators representing a subsequence of elements occurring in seq. This allows the use of logic references to iterators as arguments where the iterator type is Seq::iterator.

Note that arguments of type lref<ConvertibleToT> are not supported currently. Similarly when using iterator pairs, they must be iterators to a sequence of the same type. Using vector<T>::iterator pairs when seq is of type list<T> is not supported. All arguments other than the seq must be initialized at the time of evaluation.

**Exceptions:**

InvalidDeref : If any logic reference argument other than the first is an not initialized at the time of evaluation.

**Notes:**

sequence is a variadic relation. That is, its arity (number of arguments) is not predefined. The style of argument passing used in sequence is different compared to the traditional C style techniques used in standard variadic functions like printf and scanf. Each argument to sequence is surrounded by a ( ) pair. Thus the syntax for passing 4 arguments looks like sequence(s)(7)(8)(9) instead of sequence(s,7,8,9). This method of variadic argument passing allows relation sequence to automatically preserve full type information for each argument without additional assistance on behalf of the programmer.

The first argument represents a sequence comprising of elements described by the remaining arguments. For instance if s is a lref<list<int> >, then sequence(s)(7)(8)(9) unifies s with the sequence {7,8 ,9} . Argument s may or may not be initialized. If s is **not** initialized, it will be assigned a list<int> containing elements 7,8 and 9 in that order. If s is initialized, it will be tested to see if it contains the exactly the three elements 7, 8 and 9 in order. The first argument must be logic reference to any sequence type. The remaining arguments can be classified into two kinds. The first kind is values representing individual elements in the sequence. The second kind is a sequence representing a span of elements to need to appear in the first argument. This is specified using either iterator pairs or a logic reference to sequence. For example, if li is a list<int> then sequence**(**s**)(**li.begin(),li.end()**)** will unify s with all elements in li. Both kinds of arguments may appear in any order and can be logic references or regular types:

In many cases, the flexibility and brevity provided by variadic arguments in sequence may not be needed. In such situations, relations eq and eq\_seq provide more light weight and efficient alternatives.

**Examples:**

// 1) compare sequence with {3,4,5}

lref<vector<int> > vi = /\* {3, 4, 5} \*/;

lref<int> li = 4;

assert( sequence**(**vi**)(**3**)(**li**)(**5**)() )**; // vi == {3,4,5}

// 2) generate sequence {3,4,5}

lref<vector<int> > s; // not initialized

sequence**(**s**)(**3**)(**4**)(**5**)()**; // s = {3,4,5}

// 3) test for empty sequence

list<int> emptyList;

assert( sequence<list<int> >**(**emptyList**)()** )

// 4) Iterator pairs : generate sequence comprised of values in vi followed by 4

lref<vector<int> > s;

vector<int> vi = /\* {1,2,3} \*/ ;

int four = 4;

sequence**(**s**)(** vi.begin(),vi.end() **)(**four**)()**; // s = {1,2,3,4}

// 5) Iterator pairs : generate sequence using lref<iterator>

lref<vector<int> > s; // not initialized

vector<int> vi = /\* {1,2,3} \*/ ;

lref<vector<int>::iterator> b,e;

relation r = begin<vector<int> >(vi,b)

&& end<vector<int> > (vi,e)

&& sequence**(**s**)(**b,e**)**; // s = {1,2,3}

// 6) Simple containers comparison

lref<vector<int> > s; // not initialized

vector<int> vi = /\* {1,2,3} \*/ ;

lref<vector<int>::iterator> b,e;

relation r = sequence**(**s**)(**vi.begin(),vi.end()**)**; // s = {1,2,3}

// unification with pair of iterators can be also be done with

// the more light weight but less flexible relation eq\_seq:

relation r2 = **eq\_seq(**s**,**vi.begin(),vi.end()**)**

// simplest way to unify containers directly is to use eq:

relation r3 = **eq(**s,vi**)**;

**Also refer to:**

eq\_seq, eq, item, getValues, size, begin, end, head, tail, head\_tail, head\_n, tail\_n, head\_n\_tail.

**size relation**

template<typename Cont>

Size\_r<Cont> **size**(lref<Cont>& cont\_, lref<typename Cont::size\_type> sz)

**Declarative reading**: Size of container cont\_ is sz.

**Template Parameters:**

Cont : Must satisfy requirements of standard C++ containers [$23.1].

**Parameters:**

cont\_ : [in] Container whose size is to be determined.

sz : [in/out] Size of cont\_.

**Exceptions:**

InvalidDeref : If cont\_ is not initialized at the time of evaluation.

**Notes:**

Size of cont\_ is determined by invoking its size member function.

**Example**:

lref<vector<int>::size\_type> sz;

vector<int> v = vector<int> ();

if( size(v, sz)**()** );

cout << \*sz; // prints 0

**Also refer to:**

empty, not\_empty.

**tail relation**

template<typename Seq, typename TailSeq>

Tail\_r<Seq,TailSeq> **tail**( lref<Seq>& seq\_, lref<TailSeq>& t )

**Declarative reading**: Tail of seq\_ is t.

**Template Parameters:**

Seq: Must satisfy the requirements of standard C++ sequences [$23.1.1].

TailSeq : Must satisfy requirements of standard C++ sequences [$23.1.1].

**Parameters:**

seq\_ : [in] Container whose tail is of interest.

t : [in/out] The tail sequence of cont\_.

**Exceptions:**

InvalidDeref : If seq\_ is not initialized at the time of evaluation.

**Notes:**

The tail of a sequence comprises of all elements in sequence except for the first one (i.e. the head) .

**Also refer to:**

head, head\_n, tail\_n, head\_tail, head\_n\_tail, next, prev.

**tail\_n relation**

template<typename Seq, typename TailSeq>

relation **tail\_n**( lref<Seq>& seq\_

, lref<typename TailSeq::size\_type> n

, lref<TailSeq>& t )

**Declarative reading**: t contains last n items from sequence seq\_.

**Template Parameters:**

Seq : Must satisfy requirements of standard C++ sequences [$23.1.1].

TailSeq : Must satisfy requirements of standard C++ sequences [$23.1.1].

**Parameters:**

seq\_ : [in] Sequence whose last n elements are of interest.

n : [in/out] Number of items in t. 0 <= n <= size of seq\_.

t : [in/out] Sequence containing copies of last n element from seq\_.

**Exceptions:**

IndexOutOfBounds: If size of seq\_ is less than n\_.

InvalidDeref : If seq\_ or n\_ is not initialized at the time of evaluation.

**Notes:**

t and seq\_ may be of different types. t is a copy of the elements comprising the tail of seq\_.

**Also refer to:**

head, head\_n, tail, head\_tail, head\_n\_tail.

## Iterators

**begin relation**

template<typename Cont>

Begin\_r<Cont> **begin**( lref<Cont>& cont\_

, lref<typename Cont::iterator> iter)

**Declarative reading**: Iterator pointing to the start of container cont\_ is iter.

**Template Parameters:**

Cont : Must satisfy requirements of standard c++ containers [$23.1].

**Parameters:**

cont\_: [in] Container whose begin iterator is to be determined.

iter : [in/out] iterator to the beginning of cont\_.

**Exceptions:**

InvalidDeref : If cont\_ is not initialized at the time of evaluation.

**dereference relation**

relation **dereference**(lref<Itr> pointer\_

, lref<typename detail::Pointee<Itr>::result\_type> pointee)

**Declarative reading**: Dereferencing pointer\_ yields pointee.

**Template Parameters:**

Itr : Either a pointer or iterator type or a logic reference to a pointer or iterator.

**Parameters:**

pointer\_ : [in] A pointer or iterator to be dereferenced.

pointee : [in/out] If pointee is initialized, it will be compared with the value obtained by dereferencing pointer\_ using operator==. If not initialized, pointee will be assigned the value obtained by dereferencing pointer\_.

**Notes**:

Relation dereference is used for obtaining an lref<T> from an lref<T\*>. This is useful when iterating over containers and streams or simply working with pointers in a relational fashion.

**Examples:**

// 1) dereferencing lref<int\*> to obtain lref<int>.

int three=3;

lref<int\*> lp3=&three;

lref<int> l;

relation r = dereference(lp3,l) && write(l);

r(); // prints 3

// 2) dereferencing raw pointers.

int two=2;

int\* pi= &two;

lref<int> li;

relation r = dereference(pi, li) && write(li);

r(); // prints "2"

// 3) dereferencing logic references to iterators.

lref<vector<int> > lv = vector<int>(); lv->push\_back(4);

lref<vector<int>::iterator> lItr = lv->begin();

// check if 1st element in lv 4

relation r = begin(lv, lItr) && dereference(lItr, 4);

if(r())

cout << "first element is 4";

**end relation**

template<typename Cont>

End\_r<Cont> **end**(lref<Cont>& cont\_, lref<typename Cont::iterator> iter)

**Declarative reading**: Iterator pointing to one past the end of container cont\_ is iter.

**Template Parameters:**

Cont : Must satisfy requirements of standard c++ containers [$23.1].

**Parameters:**

cont\_: [in] Container whose begin iterator is to be determined.

iter : [in/out] points to (one past) the end of the elements in cont\_. End iterator of container is obtained by invoking its end()method.

**Exceptions:**

InvalidDeref : If cont\_ is not initialized at the time of evaluation.

**item relation**

template<typename Itr>

Item\_r<Itr> **item**( lref<typename detail::Pointee<Itr>::result\_type> obj

, Itr begin\_, Itr end\_ )

template<typename Cont>

relation **item**( lref<typename Cont::value\_type> obj, lref<Cont>& cont\_ )

**Declarative reading**: obj is an item in the sequence [begin, end) or the container cont\_.

**Template Parameters:**

Itr : Can be a pointer type, an input iterator type [$24.1.1], a logic reference to a pointer, or a logic reference to an input iterator. If Itr is not a logic reference, it must support dereferencing with operator \*. Similarly, if Itr is a logic reference type, its underlying type must support dereferencing with operator \*.

Cont : Satisfies requirements of standard containers.

**Parameters:**

obj: [in/out] obj is an item in the sequence bounded by iterators begin\_ and end\_.

begin\_: [in] points to the beginning of a sequence. It must precede or be equal to end\_.

end\_: [in] points to (one past) the end of a sequence.

cont\_: [in] A standard container whose items are of interest.

**Exceptions:**

InvalidDeref : If begin\_ or end\_ is not uninitialized at the time of evaluation.

**Notes**:

Relation item is typically useful for iterating over sequences in a relational fashion. However, due to the bidirectional nature of parameter obj, it also doubles up as a facility for testing the presence of a value in a sequence. Since item works with standard iterators and pointers (or logic references to pointers and iterators) it enables easier interaction with traditional C++ code that deal with containers, streams and arrays.

**Example:**

// 1) print all values in an array

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

lref<int> val;

lref<int\*> b = arr+0, e = arr+4;

relation r = item(obj, b, e);

while(r())

cout << \* val << " "; // prints "1 2 3 4 "

// 2) print all items in 1st array that are also part of 2nd array

// (i.e intersection of two arrays)

int arr1[] = {**1**,2,**3**,4};

int arr2[] = {6,**3**,7,**1**,9};

lref<int> i;

relation r2 = item(i, arr1+0, arr1+4) && item(i, arr2+0, arr2+5);

while(r2())

cout << \*i << " "; // prints "1 3 "

// 3) Intersection of two vectors

lref<vector<int> > v1 = vector<int>(arr1+0, arr1+4);

lref<vector<int> > v2 = vector<int>(arr2+0, arr2+5);

lref<int> i, j;

relation r3 = item(i, v1) && item(j, v2) && predicate(i==j);

while(r3())

cout << \*i << " "; // prints "1 3 "

In the second example the first call to item is responsible for generating a value for i from arr1 and the second call to item then tests if that value is part of arr2. The third example which works on vectors is semantically same as the second although expressed slightly differently.

**next relation**

template<typename T>

relation **next**(lref<T> curr\_, lref<T> n)

template<typename T>

relation **next**(T curr\_, lref<T> n)

template<typename T>

relation **next**(T curr\_, const T& n)

**Declarative reading**: Next of curr\_ is n.

**Template Parameters:**

T : Must support prefix increment operator.

**Parameters:**

curr\_ : [in] value preceding n. This must be initialized at the time of evaluation.

n : [in/out] value following curr\_. i.e ++curr\_.

**Exceptions:**

InvalidDeref : If curr\_ is a lref and is not initialized at the time of evaluation.

**Notes:**

Relation next is useful for incrementing both values and iterators. The second and third overloaded versions provide slightly optimized implementation for cases when one or both of the arguments is not a logic reference type. More importantly they simplify syntax for user code by not requiring explicit specification of the template parameter when arguments involve a mix of types lref<T> and T. Relation next generates only one solution.

**Examples:**

The following relation generates one item in i at a time in the sequence bounded by iterators b\_ and e\_. By initializing argument i to a value, this relation could be instead used to test if a particular value is present in the sequence.

relation itemsIn(lref<int\*> b\_, lref<int\*> e\_, lref<int> i) {

lref<int\*> n;

return eq(b\_,e\_) // stop if b\_==e\_

^ **(**dereference(b\_,i) || next(b\_,n) && recurse(&itemsIn, n, e\_, i) **)**;

}

For a more generalized version of itemsIn refer to documentation of relation item.

**Also refer to:**

prev, inc, dec, head, tail, item

**prev relation**

template<typename T>

relation **prev**(lref<T> curr\_, lref<T> p)

template<typename T>

relation **prev**(T curr\_, lref<T> p)

template<typename T>

relation **prev**(T curr\_, const T& p)

**Declarative reading**: Previous of curr\_ is p.

**Template Parameters:**

T : Must support prefix decrement operator.

**Parameters:**

curr\_ : [in] value succeeding p. This must be initialized at the time of evaluation.

p : [in/out] value preceding curr\_. i.e –-curr\_.

**Exceptions:**

InvalidDeref : If curr\_ is a lref and is not initialized at the time of evaluation.

**Notes:**

Relation prev is useful for decrementing both values and bidirectional iterators. The second and third overloaded versions provide slightly optimized implementation for cases when one or both of the arguments is not a logic reference type. More importantly they simplify syntax for user code by not requiring explicit specification of the template parameter when arguments involve a mix of types lref<T> and T. Relation prev generates only one solution.

**Also refer to:**

next, inc, dec, head, tail, item

## Predicates

**Boolean relation**

class Boolean : public Coroutine {

explicit Boolean(bool value);

bool operator ()(void);

};

**Brief Description**: First evaluation succeeds only if return value is true, and all subsequent evaluations fail (i.e. return false).

**Parameters:**

value: The (true/false) value to be returned on first evaluation of Boolean.

**Returns:**

If value is true, returns true on first evaluation and false otherwise. All subsequent evaluations return false.

**Exceptions:**

None.

**Notes**: This relation is useful for creating simple predicate relations from boolean values or expressions that can be eagerly evaluated.

**Example**:

int num;

cin >> num;

relation r = ( Boolean(num<5) && write("value < 5") )

^ write("value >= 5");

**Also refer to:**

True, False.

**False relation**

struct False {

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

};

**Brief Description**: Always fails.

**Returns:**

Always returns false.

**Exceptions:**

None.

**Also refer to:**

Boolean, True.

**True relation**

class True : public Coroutine {

True(); // succeed once

explicit True(unsigned long n); // succeed once ‘n’ times

bool operator ()(void);

};

**Brief Description**: Succeed n times.

**Returns:**

First evaluation returns true, and all subsequent evaluations return false.

**Exceptions:**

None.

**Also refer to:**

Boolean, False.

**predicate relation**

// overloads for function objects

template<typename Pred>

Predicate0\_r<Pred>

**predicate**(Pred pred)

template<typename Pred, typename A1>

Predicate1\_r<Pred,A1>

**predicate**(Pred pred, const A1& a1\_\_)

template<typename Pred, typename A1, typename A2>

Predicate2\_r<Pred,A1,A2>

**predicate**(Pred pred, const A1& a1\_, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to f

// overloads for function pointers

template<typename R>

Predicate0\_r<R(\*)(void)>

**predicate**(R(\* pred)(void))

template<typename R, typename P1, typename A1>

Predicate1\_r<R(\*)(P1),A1>

**predicate**(R(\* pred)(P1), const A1& a1\_)

template<typename R, typename P1, typename P2, typename A1

, typename A2>

Predicate2\_r<R(\*)(P1,P2),A1,A2>

**predicate**(R(\* pred)(P1,P2), const A1& a1\_, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to f

**Declarative reading**: pred(a1,..,aN) is true.

**Template Parameters:**

Pred : A function or function object that takes up to 6 arguments. Return type must be bool or any other type convertible to bool.

R: Return type of the function pointer. Must be bool or a type convertible to bool.

P*n* : Type of nth parameter of function pointer. Can be a POT or lref.

A*n* : Type of the nth argument to being passed to the function or function object. Can be a POT or lref whose effective type is convertible to the corresponding parameter type in Pred.

**Parameters:**

pred : Function or function object which returns bool and takes up to 6 parameters.

a*N* : [in] Argument (POT or lref) at position *N* whose effective value will be passed to pred.

**Notes:** Relation predicate is an adaptor relation used for treating regular functions with return type bool, as relations. It evaluates successfully, at most once, if pred returns true. ILEs are often used as arguments to predicate to create simple anonymous relations directly inline, thus reducing the need to declare named predicate functions. For working with predicate member functions, use predicate\_mf.

**Examples:**

Searching for even numbers in an array by adapting the predicate function isEven.

bool isEven(int num) {

return num%2 == 0;

}

int nums[] = {4,3,9,8,15};

relation evenNums = item(n, nums+0, nums+5) && predicate(isEven,n);

while(evenNums())

cout << \*n << " ";

Searching for even numbers with ILEs.

int nums[] = {4,3,9,8,15};

relation evenNums = item(n, nums+0, nums+5) && predicate(n%2==0);

while(evenNums())

cout << \*n << " ";

**Also refer to:**

predicate\_mf.

**predicate\_mf relation**

// Support for non-const member functions (with upto 6 arguments)

template<typename R, typename Obj>

MemPredicate0\_r<Obj,R(Obj::\*)(void)>

**predicate\_mf**(lref<Obj>& obj\_, R(Obj::\* mempred)(void) )

template<typename R, typename P1, typename Obj, typename A1>

MemPredicate1\_r<Obj,R(Obj::\*)(P1),A1>

**predicate\_mf**(lref<Obj>& obj\_, R(Obj::\* mempred)(P1), const A1& arg1)

template<typename R, typename P1, typename P2, typename Obj

, typename A1, typename A2>

MemPredicate2\_r<Obj,R(Obj::\*)(P1,P2),A1,A2>

**predicate\_mf**(lref<Obj>& obj\_, R(Obj::\* mempred)(P1,P2), const A1& arg1

, const A2& arg2)

.. additional overloads supporting upto 6 arguments to mf

// Support for const member functions (with upto 6 arguments)

template<typename R, typename Obj>

MemPredicate0\_r<Obj,R(Obj::\*)(void) const>

**predicate\_mf**(lref<Obj>& obj\_, R(Obj::\* mempred)(void) const)

template<typename R, typename P1, typename Obj, typename A1>

MemPredicate1\_r<Obj,R(Obj::\*)(P1) const,A1>

**predicate\_mf**(lref<Obj>& obj\_, R(Obj::\* mempred)(P1) const

, const A1& arg1)

template<typename R, typename P1, typename P2, typename Obj

, typename A1, typename A2>

MemPredicate2\_r<Obj,R(Obj::\*)(P1,P2) const,A1,A2>

**predicate\_mf**(lref<Obj>& obj\_, R(Obj::\* mempred)(P1,P2) const

, const A1& arg1, const A2& arg2)

.. additional overloads supporting upto 6 arguments to mf

**Declarative reading**: obj\_.pred(a1,..,aN) is true.

**Template Parameters:**

Obj : A type whose member function is to be treated as a relation.

R: Return type of the member function pointer. Must be bool or a type convertible to bool.

P*n* : Type of nth parameter of member function. Can be a POT or lref.

A*n* : Type of the nth argument to being passed. Can be a POT or lref whose effective type is convertible to the corresponding P*n*.

**Parameters:**

obj\_ : [in] Object on which member predicate function pointed to by mpred will be invoked. This argument must be a logic reference. This restriction ensures methods are invoked on the actual argument and not on a copy of obj\_.

mempred : Address of predicate member function to be treated as a relation.

arg*N* : [in] The Nth argument to be passed to mempred. Effective value of arg*N* is passed to mpred.

**Notes:** Relation predicate\_mf is an adaptor relation used for treating predicate member functions (having up to 6 parameters) as relations. It evaluates successfully, at most once, if pred returns true. For working with non-member predicate functions, use predicate\_f. The overloads are designed to eliminate the need for a static\_cast on mempred even in the face of overload ambiguities. Refer to examples in eval\_mf for more details on this.

**Examples:**

Counting empty lines in a file.

lref<list<string> > lines = readFromFile(..);

lref<string> line;

relation r = item(line, lines) && predicate\_mf(line,&string::empty);

int count=0;

while(r())

++count;

cout << count << " empty lines found in file.";

**Also refer to:**

predicate\_f, predicate\_mem, eq\_mf.

**predicate\_mem relation**

template<typename Obj, typename MemberT>

Predicate\_mem\_r<Obj, MemberT>

predicate\_mem(lref<Obj>& obj\_, MemberT Obj::\* mem)

**Declarative reading**: (\*obj\_).\*mem is true.

**Template Parameters:**

Obj : Any type which whose member variable is to be accessed.

MemberT : Type of the data member to be accessed. This is should be either bool or a type convertible to bool.

**Parameters:**

obj\_ : [in] The object whose data member is to be accessed. This argument must be a logic reference. This restriction ensures methods are invoked on the actual argument and not on a copy of obj\_.

mem : Pointer to a member variable.

**Notes:**

Relation predicate\_mem is used for checking the value of a boolean member variable. The relation succceds if the member variable’s value is true (or convertible to true). This relation succeeds at most once.

## Other

**dec relation**

template<typename T>

Dec\_r<T> **dec**(lref<T>& value\_);

**Declarative reading**: *value\_* is decremented.

**Template Parameters:**

T : It may be a logic reference type or a regular type. If T is a logic reference type then its underlying type (i.e. T::result\_type) must support prefix operator --. If T is not a logic reference type, it must support the prefix operator --.

**Parameters:**

value\_ : [in & out] A logic reference whose value is to be decremented. value\_ does not have to be initialized when dec is invoked but must be initialized at the time when dec is evaluated.

**Notes:** Thisrelation evaluates successfully only once. On successful evaluation value\_ will be decremented. Any further attempt to evaluate this relation will restore the original value into value\_.

**Example:**

lref<int> i;

relation r = dec(i); // 'i' need not be initialized at this point

i=2; // but must be initialized before dec is evaluated

while(r())

cout << \*i << " "; // prints 1

cout << \*i << " "; // prints 2

relation r2 = dec(3); // Compiler error. Argument must be a lref

Here, r is evaluated twice by the while loop. First evaluation attempt causes i to be decremented and evaluation succeeds. The second attempt at evaluation fails and the original value 2 is restored into i.

**Also refer to:**

inc, next, prev.

**defined relation**

template<typename T>

Defined\_r<T> **defined**(const lref<T>& r\_ )

**Declarative reading**: r\_ is initialized with a value.

**Template Parameters:**

T: Any type.

**Parameters:**

r\_ : [in] The logic reference to be tested for initialization.

**Notes:**

defined is a relational wrapper on the lref::defined method. Leaving r\_ uninitialized does not lead to generation of values for it. This relation merely invokes the defined method on r\_ when evaluated the first time. If defined returns true then evaluation succeeds, and fails otherwise. All subsequent evaluations will be unsuccessful.

**Example:**

lref<int> li=2;

//if li is initialized print its value

//otherwise print "not initialized"

relation r = ( defined(li) && write(li) )

^ write("not initialized") ;

r();

**Also refer to:**

defined.

**inc relation**

template<typename T>

Inc\_r<T> **inc**(lref<T>& value\_);

**Declarative reading**: *value\_* is incremented.

**Template Parameters:**

T : It may be a logic reference type or a regular type. If T is a logic reference type then its underlying type (i.e. T::result\_type) must support prefix ++ operator. If T is not a logic reference type, it must support the prefix ++ operator.

**Parameters:**

value\_ : [in & out] A logic reference whose value is to be incremented. value\_ does not have to be initialized when inc is invoked but must be initialized at the time when inc is evaluated.

**Notes:** Thisrelation evaluates successfully only once. On successful evaluation value\_ will be incremented. Any further attempt to evaluate this relation will restore the original value into value\_.

**Example:**

lref<int> i;

relation r = inc(i); // 'i' need not be initialized at this point

i=2; // but must be initialized before inc is evaluated

while(r())

cout << \*i << " "; // prints 3

cout << \*i << " "; // prints 2

relation r2 = inc(3); // Compiler Error. Argument must be a lref

Here, inc(i) is evaluated twice by the while loop. First evaluation attempt causes i to be incremented and evaluation succeeds. The second attempt restores the original value 2 into i and evaluation fails causing the while loop to terminate.

**Also refer to:**

dec, next, prev.

**defined relation**

template<typename T>

Defined\_r<T> **defined**(const lref<T>& r\_ )

**Declarative reading**: r\_ is initialized with a value.

**Template Parameters:**

T: Any type.

**Parameters:**

r\_ : [in] The logic reference to be tested for initialization.

**Notes:**

defined is a relational wrapper on the lref::defined method. Leaving r\_ uninitialized does not lead to generation of values for it. This relation merely invokes the defined method on r\_ when evaluated the first time. If defined returns true then evaluation succeeds, and fails otherwise. All subsequent evaluations will be unsuccessful.

**Example:**

lref<int> li=2;

//If li is initialized print its value

// otherwise print "not initialized"

relation r = ( defined(li) && write(li) )

^ write("not initialized") ;

r();

**Also refer to:**

defined.

**error relation**

template<typename ExceptionType>

Error<ExceptionType> **error**(const ExceptionType& err)

// Support for const char\* arg

template<typename ExceptionType>

Error<const ExceptionType\*> **error**(const ExceptionType\* err)

**Declarative reading**: Throw exception err.

**Template Parameters:**

ExceptionType: Type of the exception to be thrown.

**Parameters:**

err : The object to be thrown as an exception.

**Notes:**

On evaluation, this relation executes the expression throw err.

**Example:**

lref<int> li= … ;

//If li is initialized, print its value

// otherwise throw exception "not initialized"

relation r = ( defined(li) && write(li) ) ^ error("not initialized") ;

r();

**Also refer to:**

write, pause.

**eval relation**

// overloads for function objects

template<typename Func, typename A1>

Eval\_r1<Func,A1>

**eval**(Func f, const A1& a1\_)

template<typename Func, typename A1, typename A2>

Eval\_r2<Func,A1,A2>

**eval**(Func f, const A1& a1\_, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to f

// overloads for function pointers

template<typename R>

Eval\_r0<R(\*)(void)>

**eval**(R(\* f)(void))

template<typename R, typename P1, typename A1>

Eval\_r1<R(\*)(P1),A1>

**eval**(R(\* f)(P1), const A1& a1\_)

template<typename R, typename P1, typename P2, typename A1

, typename A2>

Eval\_r2<R(\*)(P1,P2), A1,A2>

**eval**(R(\* f)(P1,P2), const A1& a1\_, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to f

**Declarative reading**: Evaluate the function or function object f.

**Template Parameters:**

Func : A function object type that takes up to 6 arguments. Func must define a member typedef result\_type indicating its result type.

R: Return type of the function pointer.

P*n*: Type of the Nth parameter of function pointer. Can be an lref or POT. A*N* should be either same as or convertible to the corresponding P*n*.

A*n* : Type of the Nth argument to being passed. Can be a POT or lref whose effective type is convertible to the corresponding parameter type in Func.

**Parameters:**

f : A function pointer or function object that is to be invoked when the relation is evaluated.

a*N\_* : [in] The Nth argument to be passed to f. Effective value of a*N* is passed to f. It can be an lref or POT.

**Exceptions:**

InvalidDeref : If any argument aN is an lref and is not initialized at the time of evaluation.

Any exception thrown by f.

**Notes:** This relation can be used to execute imperative tasks, defined as a regular function of function object, during the evaluation of relations. Functions and function objects with up to 6 arguments are supported. Note that any side effects induced by f will not be undone during backtracking. Hence eval should be used with care, ensuring that it does not interfere with the correct evaluation of other relations by modifying objects that are shared with other relations. eval always succeeds once. Note that the value returned (if any) by f is not accessible. Consider using eq\_f if access to return value is required.

**Example:**

//1) Print array items using eval

void print(int x) {

cout << x << " ";

}

lref<int> x;

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

relation r = item(x,a,a+3) && eval(print,x);

while(r());

//2) Using an ILE instead of print()

relation r2 = item(x,a,a+3) && eval(ref(cout)<<x);

while(r());

The signature of eval\_f is designed to eliminate the need for a static\_cast on the f argument in user code, even in the presence of overload ambiguities. For example consider using eval\_f on the following type with a function having two overloads differing in arity (i.e. number of parameters).

int add(int i, int j) {

return i+j;

}

int add(int i, int j, int k) {

return i+j+k;

}

// messy static\_cast works but not needed eval\_f(static\_cast<int(\*)(int,int,int)>(add), 1, 2, 3)();

// equivalent simpler syntax

eval\_f(add, 1, 2, 3)();

This more direct syntax works when the overloads differ in arity. When the overloads have the same arity and differ in the parameter types we can still avoid a static\_cast by specifying only the return type and parameter types of the member function as follows.

int add(int i, int j) {

return i+j;

}

double add(double i, double j) {

return i+j;

}

// messy static\_cast works but not needed

eval\_f(static\_cast<int(\*)(int,int)>(add), 1, 2)();

// equivalent simpler syntax

eval\_f<int,int,int>(add, 1, 2)();

**Also refer to:**

eval\_mf, predicate, predicate\_mf, eq\_f, eq\_mf.

**eval\_mf relation**

// Overloads for non-const member functions

template<typename R, typename Obj>

Eval\_mf\_r0<Obj,R(Obj::\*)(void)>

**eval\_mf**(lref<Obj>& obj\_, R(Obj::\*mf)(void) )

template<typename R, typename P1, typename Obj, typename A1>

Eval\_mf\_r1<Obj,R(Obj::\*)(P1),A1>

**eval\_mf**(lref<Obj>& obj\_, R(Obj::\* mf)(P1), const A1& a1\_)

template<typename R, typename P1, typename P2, typename Obj

, typename A1, typename A2>

Eval\_mf\_r2<Obj,R(Obj::\*)(P1,P2),A1,A2>

**eval\_mf**(lref<Obj>& obj\_, R(Obj::\* mf)(P1,P2), const A1& a1\_

, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to mf

// Overloads for const member functions

template<typename R, typename Obj>

Eval\_mf\_r0<Obj,R(Obj::\*)(void) const>

**eval\_mf**(lref<Obj>& obj\_, R(Obj::\*mf)(void) const)

template<typename R, typename P1, typename Obj, typename A1>

Eval\_mf\_r1<Obj,R(Obj::\*)(P1) const,A1>

**eval\_mf**(lref<Obj>& obj\_, R(Obj::\* mf)(P1) const, const A1& a1\_)

template<typename R, typename P1, typename P2, typename Obj

, typename A1, typename A2>

Eval\_mf\_r2<Obj,R(Obj::\*)(P1,P2) const,A1,A2>

**eval\_mf**(lref<Obj>& obj\_, R(Obj::\* mf)(P1,P2) const, const A1& a1\_

, const A2& a2\_)

.. additional overloads supporting upto 6 arguments to mf

**Declarative reading**: Evaluate member function mf on object obj\_.

**Template Parameters:**

Obj : A type whose member function is to be invoked.

R : Return type of the member function.

P*n* : Type of the nth parameter of member function.

A*n* : Type of the nth argument to being passed. Can be a POT or lref whose effective type is convertible to the corresponding parameter type P*n*.

**Parameters:**

obj\_ : [in] Object on which member function pointed to by mf will be invoked. This argument must be a logic reference. This restriction ensures methods are invoked on the actual argument and not on a copy of obj\_.

mf : Pointer to a member function that is to be invoked when this relation is evaluated.

a*N\_* : [in] The Nth argument to be passed to mf. Effective value of a*N* is passed to mf. Thus it can be an lref or POT.

**Exceptions:**

InvalidDeref : If any argument a*N* is a lref and is not initialized at the time of evaluation.

Any exception thrown by mf.

**Notes:** This relation can be used to execute imperative tasks defined in member functions during the evaluation of relations. Member functions with up to 6 arguments are supported. Note that any side effects induced by mf will not be undone during backtracking. Hence eval\_mf should be used with care, ensuring that it does not interfere with the correct evaluation of other relations by modifying objects that are shared with other relations. eval\_mf always succeeds once. Note that the value returned (if any) by mf is not accessible. Consider using eq\_mf if access to return value is required.

**Examples:**

The signature of eval\_mf is designed to eliminate the need for a static\_cast[[2]](#footnote-2) on the mf argument in user code, even in the presence of overload ambiguities. For example consider using eval\_mf on the following type with a member function having two overloads differing in arity (i.e. number of parameters).

struct calculator {

int add(int i, int j) {

return i+j;

}

int add(int i, int j, int k) {

return i+j+k;

}

};

lref<calculator> lc = calculator();

eval\_mf(lc

, static\_cast<int(calculator::\*)(int,int,int)>(&calculator::add)

, 1, 2, 3)(); // messy static\_cast works but not needed

eval\_mf(lc, &calculator::add, 1, 2, 3)(); // equivalent simpler syntax

This more direct syntax works when the overloads differ in arity. When the overloads have the same arity and differ in the parameter types we can still avoid a static\_cast by specifying only the return type and parameter types of the member function as follows.

struct calculator {

int add(int i, int j) {

return i+j;

}

double add(double i, double j) {

return i+j;

}

};

lref<calculator> lc = calculator();

eval\_mf(lc

, static\_cast<int(calculator::\*)(int,int)>(&calculator::add)

, 1, 2)(); // messy static\_cast works but not needed

eval\_mf<int,int,int>(lc, &calculator::add, 1, 2)();// simpler syntax

Relation eval\_mf also safe to use with virtual and pure virtual functions.

struct Shape {

virtual void draw()=0;

virtual ~Shape(){}

};

struct Circle : public Shape {

virtual void draw() { cout << "Circle"; }

};

lref<Shape> lc = Circle();

eval\_mf(lc, &Shape::draw)();// calls Circle::draw()

**Also refer to:**

eval\_f, predicate, predicate\_mf, eq\_f, eq\_mf.

**pause relation**

template<typename T>

Pause\_r<T> **pause**(lref<T>& msg)

template<typename T>

Pause\_r<T> **pause**(const T& msg)

template<typename T>

Pause\_r<const T\*> **pause**(const T\* msg)

**Declarative reading**: Print msg to std::cout and wait for key press to continue.

**Template Parameters:**

T : Type of object to be printed. Should support the expression cout<<msg where msg is of type T.

**Parameters:**

msg: [in] The object to be printed.

**Exceptions:**

InvalidDeref : If msg is an lref and not initialized at the time of evaluation.

**Notes:**

On evaluation, the relation prints the object to cout and then performs a cin.ignore() to waiting for user to press the ENTER key. This relation is useful for debugging purposes when one wishes to manually step through an execution.

**Examples:**

// print each number in range and wait for keypress each time

lref<int> li;

relation r = range(li,0,3) && pause(li);

while(r());

**Also refer to:**

write.

**range relation**

template<typename T>

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

template<typename T>

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

template<typename T>

Range\_Step\_r<T> **range**(lref<T> val, T min\_, T max\_, T step\_)

template<typename T>

Range\_Step\_r<T> **range**(lref<T> val, lref<T> min\_, lref<T> max\_, lref<T> step\_)

**Declarative reading**: val is >= min\_ and <= max\_.

**Template Parameters:**

T : For overloads *without* the step\_ parameter T must support <,== and prefix ++.

For overloads *with* the step\_ parameter T must support <, == and +=.

**Parameters:**

val: [in/out] val lies within the range (min\_, max\_).

min\_: [in] Specifies an inclusive lower bound that is less than or equal to max\_.

max\_: [in] Specifies an inclusive upper bound that is greater than or equal to min\_.

step\_: [in] Specifies an increment to use (only) when generating values for val\_. This is not used when checking if val\_ is in the inclusive range.

**Exceptions:**

InvalidDeref : If min\_ or max\_ is not initialized at the time of evaluation.

**Notes:**

If min\_ is greater than max\_ the range is considered empty and the relation will never succeed.

**Examples:**

// 1) print all values in the inclusive range (0,3)

lref<int> li;

relation r = range(li,0,3);

while(r()) // prints "0 1 2 3 "

cout << \*li << " ";

// 2) print alternate values in the inclusive range (0,10)

lref<int> li;

relation r = range(li,0,10,2);

while(r()); // prints "0 2 4 6 8 10"

cout << \*li << " ";

// 3) check if 12 is in the inclusive range (3,19)

relation r = range(12,3,19);

if(r())

cout << "Yes.";

// 4) empty range (i.e. min\_ > max\_ )

relation r = range(i,10,2);

**Also refer to:**

item, eq\_seq.

**repeat relation**

template<typename T>

Repeat\_r<T>

**repeat**(lref<T>& val\_i, unsigned int count\_i, lref<T>& r)

template<typename T>

Repeat\_r<T> **repeat**(T val\_i, unsigned int count\_i, lref<T>& r)

**Declarative reading**: r\_ is not initialized with a value.

**Template Parameters:**

T : Any type that satisfies the requirements of lref.

**Parameters:**

val\_i : [in] The value to be repeated.

count\_i : [in] The number of times to repeat.

r : [out] Value of val\_i will be repeated in r.

**Notes:**

repeat succeeds count\_i times, each time producing the original value of val\_i in r. Assignment of

Note that the value of val\_i is assigned to r only once on first evaluation of repeat. Thus any changes to the value of val\_i in between evaluations of repeat will not reflect in r. Also r is not reassigned the original value of val\_i on each evaluation. If r is found to be already initialized on first evaluation, this value will be memorized prior to being overwritten and on final evaluation of repeat this value will be restored into r.

**Example:**

// repeat ‘1’ three times in j

lref<int> j;

int times=3;

relation r = repeat(1,times,j);

while( r() )

cout << s << " ";

**unique relation**

template<typename T>

Unique\_r<T> **unique**(lref<T> item\_)

**Declarative reading**: item\_ has not been seen before.

**Parameters:**

item\_ : [in] The value to be tested for uniqueness. item\_ must be initialized at the time of evaluation.

**Notes:**

Duplicate results are commonly observed in logic programming. Relation unique is useful in filtering out duplicates from the results that are generated from other relations. An evaluation of unique succeeds only if it has encountered the current value of item\_ for the first time. Internally unique maintains a set of items of type T. Each time evaluation is triggered, it consults this set to determine if item\_ has been noticed before, if not item\_ is added to the set. Note that backtracking does not cause the relation to forget which items have been observed before. Its semantics depends on this “memory”.

**Exceptions:**

InvalidDeref : If item\_ is not initialized at the time of evaluation.

**Example:**

// print items in arr[] after filtering out duplicate ocurrences

int arr[] = {0, 1, 2, 3, 3, 2};

lref<int> i;

relation r = item(i, arr+0, arr+5) && unique(i);

while(r())

cout << \*i << " ";

**unique\_f relation**

template<typename FuncObj>

Unique\_f\_r<FuncObj> **unique\_f**(FuncObj f)

**Declarative reading**: Value returned by f() has not been seen before.

**Template Parameters:**

FuncObj : A function object that does not take any parameters. It must provide a member typedef result\_type stating the return type of its member operator()(void). This must be a function object and cannot be a regular function type.

**Parameters:**

f : The function object whose return value has not been seen before.

**Notes:**

This relation is similar to relation unique except that its argument is a function object whose return value is used to perform the uniqueness check. ILEs are often useful as arguments to unique\_f.

**Exceptions:**

Any exception thrown by f().

**Example:**

// if i is an item in arr1 and j is an item in arr2

// print all pairs of i and j such that i\*j is unique

int arr1[] = {0, 1, 2, 3, 3, 2};

int arr2[] = {3, 2, 1, 6, 3, 1};

lref<int> i, j;

int expected=0;

relation r = item(i, arr1+0, arr1+5) && item(j, arr2+0, arr2+5)

&& unique\_f(i\*j);

while(r())

cout << \*i << " " << \*j << "\n";

**unique\_mf relation**

template<typename R, typename Obj>

UniqueMf\_r<R,R(Obj::\*)(void), Obj>

**unique\_mf**( lref<Obj>& obj\_, R(Obj::\*mf)(void) )

template<typename R, typename Obj>

UniqueMf\_r<R,R(Obj::\*)(void) const, Obj>

**unique\_mf**( lref<Obj>& obj\_, R(Obj::\*mf)(void) const)

**Declarative reading**: Value returned by ((\*obj\_).\*mf)() has not been seen before.

**Template Parameters:**

R : Return type of member function.

Obj : Type of the object on which the member function is to be invoked.

**Parameters:**

obj\_ : Lref to an object on which member function is to be invoked.

mf : Pointer to a member function on the object.

**Notes:**

This relation is similar to relation unique\_f except that its argument is a member function whose return value is used to perform the uniqueness check.

**Exceptions:**

InvalidDeref : If obj\_ is not initialized at the time of evaluation.

Any exception thrown by mf().

**Example:**

// Filter based on string’s length

string words[] = {"mary", "had", "a", "little", "lamb"};

lref<string> w;

relation r = item(w,words,words+5) && unique\_mf(w,&string::length);

while(r())

cout << \*w << " "; // ‘lamb’ will not be printed

**unique\_mem relation**

template<typename Obj, typename MemberT>

UniqueMem\_r<Obj, MemberT>

**unique\_mem**(lref<Obj>& obj\_, MemberT Obj::\* mem)

**Declarative reading**: (\*obj\_).\*mem has not been seen before.

**Template Parameters:**

MemberT : Type of data member.

Obj : Type of the object on whose the data member is to be accessed.

**Parameters:**

obj\_ : Lref to an object whose data member is to be accessed.

mem : Pointer to a data member on the obj\_.

**Notes:**

This relation is similar to relation unique\_mf except that its argument is a data member whose value is used to perform the uniqueness check.

**Exceptions:**

InvalidDeref : If obj\_ is not initialized at the time of evaluation.

**Example:**

struct person {

string firstName, lastName;

person (string firstName, string lastName)

: firstName(firstName), lastName(lastName)

{ }

bool operator==(const person& rhs) const {

return (firstName==rhs.firstName) && (lastName==rhs.lastName);

}

};

// Print unique last names

person people[] = { person("Roshan","Naik"), person("Runa","Naik")

, person("Harry","Potter") };

lref<person> p;

relation r = item(p,people,people+3)&& unique\_mem(p,&person::lastName);

while(r())

cout << p->lastName << " ";

# Cuts

**Introduction**

The term *cut* refers to a facility used in LP for altering the default backtracking behavior. Its primary purpose is to dynamically eliminate from consideration some candidate paths of evaluation during backtracking. By default, backtracking pursues all possible paths of evaluation even if the paths do not produce any useful results. Backtracking itself has no knowledge about which paths are likely to produce results and which will not. However, the programmer may have sufficient knowledge to determine that in certain cases, pursuing alternate paths later will be simply wasteful. For instance consider the following relation which prints the result after comparing its two arguments:

relation greaterLessEq(lref<int> n

, lref<int> cmpVal) {

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

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

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

|| predicate(n==cmpVal) && write("equal") **)**;

}

It is clear by observation that if predicate(n<cmpVal) in the first clause succeeds, both

predicate(n>cmpVal) and predicate(n==cmpVal) in the subsequent clauses will fail. Similarly if first clause fails and the second clause succeeds due to successful evaluation of predicate(n==cmpVal), the third clause can be ignored by backtracking. Thus the successful evaluation of predicate(n<cmpVal) and predicate(n>cmpVal) are two important stages in the evaluation of this relations. At each of these points we can commit to the current path of evaluation and discard all alternatives. In other words, we can “cut out” the alternative paths. We can redefine the above relation using cuts by as follows:

relation greaterLessEq2(lref<int> n

, lref<int> cmpVal) {

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

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

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

|| predicate(n==cmpVal) && write("equal") **)**;

}

This definition includes two important changes. First, we have specified cut() at each point where we are ready to commit to one path. These points are called *cut points*. Second, we have enclosed the three clauses separated by disjunction operators in a cutexpr(...). The cut() and the cutexpr() are used in conjunction to specify the point at which to commit to a path and the extent of the path we are interested in committing to. By using cut(), we specify the points at which to commit, and by using cutexpr() we indicate the extent or the scope within which the cut points take effect.

In the above example, if predicate(n<cmpVal) succeeds, backtracking will encounter cut() and consequently eliminate from consideration all alternatives available just after the opening bracket of cutexpr and up until the cut(). All alternatives available before the cutexpr and all alternatives after the cut() are left as is. So, for instance, if the first clause is rewritten as:

... predicate(n<cmpVal) && **cut()** && **(** write("lesser") || write("smaller") **)**

Here we have two alternative write clauses immediately following the cut().This cut point will not influence the choices that backtracking will make when evaluating the two write clauses. Backtracking only commits to the path starting at cutexpr and ending at the cut point.

A cut point without a surrounding cutexpr, or a cutexpr without any cut points are both meaningless. By design, such mismatched occurrences will produce compilation errors. The 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()** ...

}

Since cuts interfere with readability, their usage should be limited to cases when they have sufficiently significant effect on performance. The exclusive or operator defined over relations could be considered in many situations where cuts are applicable. Conceptually the ex-or operator is a special case of the cut facility, only more readable. In the above greaterLessEq example, we may simply replace all || operators with ^ as follows:

relation greaterLessEq(lref<int> n, lref<int> cmpVal) {

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

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

^ ( predicate(n>cmpVal) && write("greater") )

^ ( predicate(n==cmpVal) && write("equal") ) **)**;

}

Also note the use of additional brackets around each clause separated by the ^ operator. This is because operator ^ has a higher precedence than &&.

Support for cuts is provided in Castor via relation cutexpr, class cut and overloaded operators &&, || and ^.

**cutexpr relation**

template<typename ExprWithCut>

CutExpr\_r<ExprWithCut> **cutexpr**(const ExprWithCut& cut\_expr)

**Declarative reading**: n/a.

**Template Parameters:**

ExprWithCut : A type that implements member function bool exec(bool&).

**Parameters:**

cut\_expr : This is a relation expression that includes at least one cut().

**Exceptions:**

Any exception thrown by cut\_expr.

**Notes**:

Relation cutexpr provides a scope within which a cut operates. Refer to the introductory section above on Cuts.

**Also refer to:**

cut

**cut class**

**Purpose:** Introduces a cut point in a cut expression.

**Class Definition:**

class cut{};

**Notes**:

The class cut is a trivial type with no user defined members. An instance of cut is used solely to mark a cut point. Refer to the introductory section above on Cuts.

# Helper classes, functions and macros

**Coroutine class**

**Purpose:** Optional helper base class for implementing relations as classes. Enables use of macros co\_begin, co\_end, co\_yield, co\_return in derived classes.

**Class Declaration:**

class Coroutine {

public:

Coroutine() ;

};

**Notes**:

The task of implementing an arbitrary relation as a class can be simplified by using coroutine style implementation. Since C++ does not support coroutines natively, Castor provides four macros (co\_begin, co\_end, co\_yield, co\_return) that simulate the coroutine style programming model.

To define a relation class as a coroutine, we derive from custom\_relation and implement the function call operator bool operator()(void) as follows:

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

class Myrelation\_r : public custom\_relation {

lref<int> p1, p2;

public:

Myrelation\_r(lref<int> p1, lref<int> p2) : p1(p1), p2(p2)

{ }

bool operator() () {

co\_begin();

... // definition of coroutine goes here

co\_end();

}

};

Note the use of macro co\_begin to start and macro co\_end to end the body of operator(). No statements 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(). Also avoid defining local variables inside operator(), since their state will not persist across invocations of operator(). This is demonstrated in example 8 below, which is a more natural but incorrect way of implementing example 7.

**Examples:**

// 1) Simplest coroutine that never succeeds

struct Simple1 : public custom\_relation {

bool operator()(void) {

co\_begin();

co\_end();

}

};

Simple1 r;

cout << boolalpha << r() << "\n"; // prints false

cout << boolalpha << r() << "\n"; // prints false

// 2) Coroutine that succeeds once, using co\_yield

struct Simple2 : public custom\_relation {

bool operator()(void) {

co\_begin();

co\_yield(true);

co\_end();

}

};

Simple2 r;

cout << boolalpha << r() << "\n"; // prints true

cout << boolalpha << r() << "\n"; // prints false

// 3) Succeeds once, using co\_return

struct Simple3 : public custom\_relation {

bool operator()(void) {

co\_begin();

co\_return(true);

co\_end();

}

};

Simple3 r;

cout << boolalpha << r() << "\n"; // prints true

cout << boolalpha << r() << "\n"; // prints false

// 4) Succeeds twice, using co\_yield

bool operator()(void) {

co\_begin();

co\_yield(true);

co\_yield(true);

co\_end();

}

// 5) Succeeds only once

bool operator()(void) {

co\_begin();

co\_return(true);

co\_return(true); // will never be executed

co\_end();

}

// 6) Succeeds only once

bool operator()(void) {

co\_begin();

co\_yield(false);

co\_yield(true); // will never be executed

co\_end();

}

// 7) Succeeds ‘n’ times

class SimpleN : public custom\_relation {

int n, i;

public:

SimpleN(int n) : n(n), i(0)

{}

bool operator()(void) {

co\_begin();

while(i++ < n)

co\_yield(true);

co\_end();

}

};

// 8) Incorrect way of implementing relation SimpleN from above

class SimpleN : public custom\_relation {

int n;

public:

SimpleN(int n) : n(n)

{}

bool operator()(void) {

co\_begin();

for(int i=0; i<n; ++i) // ‘i’ should not be defined here

co\_yield(true);

co\_end();

}

};

// Example 9 : Relation to test/generate size of a specified string

class StrSize : public custom\_relation {

lref<string::size\_type> sz;

lref<string> str\_;

public:

// str\_ is an input only parameter, sz is in/out

StrSize(lref<string> str\_, lref<string::size\_type> sz)

: sz(sz), str\_(str\_)

{ }

bool operator() (void) {

co\_begin();

if(sz.defined())

co\_return( \*sz == str\_->size() );

sz = str\_->size();

co\_yield(true);

sz.reset(); // revert external side effects

co\_end();

}

};

cout << boolalpha << StrSize("blah",4)(); // prints true

lref<string::size\_type> sz;

StrSize("blah",sz)();

cout << \*sz; // prints 4

**Also refer to:**

predicate, co\_begin, co\_end, co\_yield, co\_return

**co\_begin macro**

#define co\_begin() ...

**Brief Description**: Used at the beginning of the method body of a coroutine. T This macro can only be used in non-static member functions and requires the enclosing class to derive (public/protected) from class Coroutine.

**Also refer to:**

Coroutine, co\_end, co\_yield, co\_return

**co\_end macro**

#define co\_end() ...

**Brief Description**: Used at the end of the method body of a coroutine. The macro also implicitly returns false to the caller. This macro can only be used in non-static member functions and requires the enclosing class to derive (public/protected) from class Coroutine.

**Also refer to:**

Coroutine, co\_begin, co\_yield, co\_return

**co\_return macro**

#define co\_return(booleanExpr) ...

**Brief Description**: Used by coroutines to return a value to caller. This macro indicates completion of the lifetime of the coroutine. All future attempts to execute the coroutine instance will return false. This macro can only be used in non-static member functions and requires the enclosing class to derive (public/protected) from class Coroutine.

**Parameters:**

booleanExpr: Any expression that evaluates to true or false. This value will be return back to the caller of the coroutine.

**Also refer to:**

Coroutine, co\_begin, co\_end, co\_yield

**co\_yield macro**

#define co\_yield(booleanExpr) ...

**Brief Description**: Used by coroutines to return a value to caller. When the argument to co\_yield evaluates to true, this macro indicates a temporary suspension of execution of the coroutine and it returns true back to the caller. Next time the coroutine is invoked, it resumes execution directly from the last co\_yield, skipping all statements preceding the co\_yield. However if the argument to co\_yield evaluates to false, it indicates the completion of the lifetime of the coroutine similar to co\_return. All future attempts to execute the coroutine instance will return false. This macro can only be used in non-static member functions and requires the enclosing class to derive (public/protected) from class Coroutine.

**Parameters:**

booleanExpr: Any expression that evaluates to true or false. This value will be return back to the caller of the coroutine.

**Also refer to:**

Coroutine, co\_begin, co\_end, co\_return

**TestOnlyRelation class**

**effective\_value function**

template <typename T>

T& effective\_value(T& obj) {

return obj;

}

template <typename T>

const T& effective\_value(const T& obj) {

return obj;

}

template <typename T>

T& effective\_value(lref<T>& obj) {

return \*obj;

}

template <typename T>

const T& effective\_value(const lref<T>& obj) {

return \*obj;

}

**Brief Description**: If t1 is a logic reference then its effective value is obtained by the expression \*t1. Effective value of any other object t2 is t2 itself.

**Template Parameters:**

T : Any type.

**Parameters:**

obj: The object whose effective value is desired.

**Returns:**

The effective value of obj.

**Exceptions:**

InvalidDeref : If obj is an uninitialized lref.

**Example**:

lref<int> li=2;

int i=3;

cout << effective\_value(li); // prints 2

cout << effective\_value(i); // prints 3

**Also refer to:**

effective\_type.

**effective\_type class (meta function)**

template<typename T>

struct effective\_type {

typedef T result\_type;

};

template<typename T>

struct effective\_type<lref<T> > {

typedef typename lref<T>::result\_type result\_type;

};

**Brief Description**: Effective type of a logic reference lref<T1> is T1. Effective type of any other type T2 is T2 itself.

**Template Parameters:**

T : Any type.

**Parameters:**

obj: The object whose effective value is desired.

**Returns:**

The effective value of obj.

**Notes:** Class effective\_type provides a single member typedef result\_type for determining the effective type of any given type.

**Example**:

effective\_type<lref<string> >::result\_type str; //str’s type is string

effective\_type<string>::result\_type str2; //str2’s type is string

**Also refer to:**

effective\_value.

**getValueCont function**

template<typename ContOfT, typename ContOfLrefT>

ContOfT **getValueCont**(const ContOfLrefT& cont)

**Brief Description**: Produces a sequence of POT values from a sequence of logic references(or pointers or iterators). For example, it can be used to obtain a vector<int> from a vector<lref<int> >. All logic references in cont must be initialized.

**Template Parameters:**

ContOfT : The type of container to be returned by the function. This type must always be explicitly specified as the compiler cannot infer a type for this. Must satisfy requirements of standard C++ containers [$23.1].

ContOfLrefT : A container of logic references (or pointers or iterators) from which values are to be extracted by dereferencing each element. Must satisfy requirements of standard C++ containers [$23.1].

**Parameters:**

cont : A sequence of logic initialized references.

**Returns:**

A sequence of values obtained by dereferencing each logic reference in cont.

**Exceptions:**

InvalidDeref : If any logic reference in cont is not initialized at the time of evaluation.

**Notes**: Time complexity is O(n), where n is the number of elements in seq.

**Example**:

list<lref<int> > lri; // list of logic refs

lri.push\_back(1); lri.push\_back(2); lri.push\_back(3);

vector<int> vi = getValues<vector<int> >(lri);

copy(vi.begin(), vi.end()

, ostream\_iterator<int>(cout," ")); // prints 1 2 3

**Also refer to:**

predicate.

**OneSolutionRelation class [deprecated: use Coroutine]**

**Purpose:** Useful as a base class when imperatively implementing relations that produce at most one solution.

**Class Definition:**

template<typename Derived>

class OneSolutionRelation {

public:

OneSolutionRelation();

bool operator() (void);

};

**Template Parameters:**

Derived : Must implement methods bool apply() and void revert().

**Notes**:

Implementing a relation using imperative techniques often involves placing the imperative code in a function object. To simplify some of the chore involved in the implementation, OneSolutionRelation may be used as a public base class of the function object. Note, this class is only useful in implementing relations that generate at most one solution. The derived function object is required to implement two methods apply and revert. OneSolutionRelation implements the bool operator() which invokes these methods from the derived type. apply is invoked when the evaluation is triggered on the relation for the first time. revert is called when the evaluation is triggered for the second time. Thereafter neither apply nor revert will be invoked, instead operator() immediately returns false to the caller. Like any other relation it returns true if it succeeds or false otherwise. On failure, the lref arguments to the relation should be left unmodified. On success, ifany of the lref arguments were modified, these changes are expected to reverted in the revert method. Note that revert will only be called if apply succeeded previously.

**Examples:**

//-------------------------------------------

//1) Succeeds once, fails thereafter

struct True : OneSolutionRelation<True> {

bool apply() {

return true; // succeed trivially

}

void revert() {

// no side effects to revert

}

};

relation r = True();

while(r()) // condition will only succeed once

cout << "success";

//-------------------------------------------

//2) relation to generate/test string sizes

class StringSize : public OneSolutionRelation<StringSize> {

lref<string::size\_type> sz;

lref<string> str\_;

bool sz\_changed;

public:

// str\_ is an input only parameter, sz is in/out

StringSize(lref<string> str\_, lref<string::size\_type> sz)

: sz(sz), str\_(str\_), sz\_changed(false)

{ }

bool apply (void) {

if(sz.defined())

return \*sz == str\_->size();

sz = str\_->size();

sz\_changed = true;

return true;

}

void revert(void) {

if(sz\_changed) {

sz.reset();

sz\_changed = false;

}

}

};

lref<string> str = "Hello";

lref<string::size\_type> sz;

relation r = StringSize(str,sz) && write(sz);

r();

**Also refer to:**

custom\_relation, predicate

1. Composing relations from regular functions requires some care, due to the lazy evaluation semantics of relations, which is in contrast to eager evaluation semantics of regular functions. [↑](#footnote-ref-1)
2. Microsoft Visual C++ 2008 compiler needs explicit resolution when there exists a const and a non-const overload for the member function. CodeGear C++ Builder 2007 compiler does not support this ability when there exists multiple overloads for the member function having the same arity. In this case *all* template arguments must be specified in addition to explicitly resolving the particular member function overload. [↑](#footnote-ref-2)