# Implementation of a Common Lisp-like type algebra system in C++ templates

Hyungrok Kim
Korea Science Academy
111 Baegyanggwanmun-ro
Busanjin-gu, Busan, Republic of Korea

February 1, 2007

## Contents



## 1 Introduction

Statically typed languages—such as Ada, C, C++, Fortran, Haskell, Java, ML, Modula, and Pascal—by definition have a built-in static type system designed to ensure that type-safety violations do not occur, though most languages provide some means to override type safety, such as type-casts and C's `void*`. Because of the built-in nature of the type system, it has been very difficult to

expand with new features; for instance, simulating object-orientation to C without a language extension is difficult and involves complex, hard-to-understand, redundant patterns (i.e. records with pointers to functions).

In contrast, the more dynamic languages, such as Lisp, Perl, Python, and Ruby, provide much more freedom. For instance, it is possible to add almost native object-orientation to Lisp entirely within Lisp itself in two pages of code, [Gra93, p. 350] and CLOS, the standardized Common Lisp Object System which is one of the most powerful object-oriented facilities available in any language, can be completely implemented within plain Common Lisp.

However, modern C++ has added a powerful facility called templates, which enables a form of compile-time metaprogramming similar in some ways to Lisp's macros. With the use of template metaprogramming, features such as multi-methods can be implemented. [Ale01b, Ch. 11]

One of the more distinctive features of Common Lisp is its expressive, flexible type declaration system, with operators for type algebra. This paper considers how Common Lisp's type system can be simulated in C++ using templates and discuss the advantages and possible drawbacks of the approach.

## 2 The Common Lisp type system

Common Lisp, like other languages in the Lisp family, is dynamically typed: an object's type is unknown at compile-time in the general case, although frequently optimizers are able to infer the type information. However, in order to aid optimizers, Common Lisp has an optional type declaration syntax that enables programmers to assert the type of a function argument, variable, etc. Note that the Common Lisp standard [INC94] doesn't require run-time checking of the programmer's declarations; if the declarations are wrong, the consequences are undefined, likely involving program crashes.

Type declarations in Common Lisp are very expressive, perhaps the most expressive among all past and current languages in existence. Common Lisp's types—*type specifiers*, in the terminology of the Standard [INC94, §4.2.3]—are divided into two classes: *atomic type specifiers* and *compound type specifiers*.

Atomic type specifiers are type specifiers composed of a single symbol, such as `integer`, `list`, and `symbol`. On the other hand, compound type specifiers are (possibly nested) lists of symbols—for instance, `(vector * 100)` for a vector containing a hundred elements of an unspecified type and `(integer -7 8)` for the type comprising all integers in the interval $[-7, 8]$, inclusive. These should be considered direct analogues of C++ templates.

However, Common Lisp also defines additional compound type specifiers that enable set-theoretic operations on types: for instance, the type `(and list symbol)` denotes the type comprising all objects that are simultaneously lists and symbols—that is, the type `null`, containing the single value `nil`. A list of Common Lisp general type operators is given in Table 1.

Note that by generalizing the notion of a type as just a set of values, the is-a relation in object-oriented programming theory is generalized to in-

| Common Lisp | Mathematics | Description |
|---|---|---|
| `(and` $A$ $B$`)` | $A \cap B$ | Intersection of two types |
| `(or` $A$ $B$`)` | $A \cup B$ | Union of two types |
| `(not` $A$`)` | $\overline{A}$ | Complement of a type |
| `(eql` $a$`)` | $\{a\}$ | Type consisting of a singleton |
| `(member` $a_1$ $a_2$ ... $a_n$`)` | $\{a_1, a_2, \ldots, a_n\}$ | A fixed set of values |
| `(satisfies` $P$`)` | $\{x \mid P(x)\}$ | Values that satisfy a predicate |
| `t` | $U$ | The type of all values |
| `nil` | $\varnothing$ | The null type |

Table 1: Common Lisp type-algebraic operators and their mathematical equivalents.

clude general subtypes, type-theoretic equivalents of subsets. For instance, it is mathematically obvious that `(and` $A$ $B$`)` $\subseteq A \subseteq$ `(or` $A$ $B$`)`, where the symbol $\subseteq$ is used to denote the is-a relation. However, it is not always clear whether a type is a subtype of another—for instance, it is not known whether `(and (satisfies oddp) (satisfies perfectp))` $\subseteq$ `nil` is true (where `oddp` and `perfectp` denote predicates for odd numbers and perfect numbers, respectively). Therefore, the Common Lisp built-in function `(subtypep` $A$ $B$`)` returns two values: one value is a Boolean value denoting whether the subtype relation is true, and the other denotes whether the previous value is valid. [INC94, §4.4]

# 3 The C++ type system

In stark contrast, the C++ type system is relatively mundane, though richer than other languages in the C family, such as C, Java, or C#. [RS05] gives a formal description of C++'s types, including inheritance, overload resolution, and templates. In this type system, the only is-a relation provided is inheritance, which can be multiple inheritance. Thus, the type system is a directed acyclic graph, in contrast to languages like Java and C#, which provide only single inheritance and so-called interfaces, i.e. abstract classes without implementation that can be multiply inherited from. [Str97, §12.2.4]

Another unique feature (among the C family) of C++ is its template system, which is rich enough to be a Turing-complete sublanguage in its own right, unlike similar but weaker generics features in Java, C#, or Ada. [Vel03] In particular, they can be viewed as a form of metafunctions—functions with types and/or compile-time constants as arguments and returning types (with the `typedef` facility), compile-time values, or even functions. [AG04]

Finally, C++ provides user-definable conversion operators, which enable another form of pseudo–is-a relation, though in this case the rules for applying user-defined conversions is rather complicated. Furthermore, the standard type conversion operators—`static_cast`, `dynamic_cast`, `reinterpret_cast`, and `const_cast`—are identical in syntax to user-definable template functions, so

custom cast operators can be defined almost natively.

# 4 Exploiting the template mechanism for type algebra

Using the template metafunction technique, "type operators" that function similarly to Lisp's `and` or `or` can be implemented.

## 4.1 Implementation of type disjunction

Type disjunction, i.e. Common Lisp's `or` compound type specifier, is perhaps the most useful among Lisp's types. In effect, it is a type that can hold two or more different, unrelated types. In other words, type disjunction is equivalent to unions, which are natively supported by C, C++, and Ada (but not Java or C#).

However, it is widely known within the C++ community that the union construct is very dangerous and hazardous. For instance, C++ unions do not natively support querying the actual type of a union value, and manual type code enumerations must be maintained. Furthermore, no safety checking whatsoever is done at run-time. This lack creates serious type-safety problems, typically resulting in undefined behavior, meaning that anything can happen. [SA04, §97]

Another commonly seen but nevertheless undesirable technique is the use of unsafe casts. In C, the cast operator `(`*type*`)` can result in a number of conversions that result in undefined behavior, such as pointer-to-integer conversion. Although simple, this technique is inconvenient because it shares the type-safety violations of the previous technique and also because the type information is not explicitly specified.

In C++, the use of C-style casts is discouraged, and the more verbose C++-style cast operators are to be used instead. [Mey95, item 2] Among the C++-style cast operators, two operators can be used to cast forcibly between incompatible types: `reinterpret_cast` will cast between any two types provided that they do not differ in their cv-qualifiers (i.e. `const` and `volatile` type qualifiers), and `const_cast` will cast between two types differing only in their cv-qualifiers. Together, they can simulate most of the functionality of the old C cast, and the C cast operator is deprecated in the C++ standard. Nevertheless, they are still unsafe. In addition, they bear the additional stigma of being unnecessarily verbose (which is intentional, as noted above).

A safer and more recommended technique is the use of templates to create *discriminated unions*—type-safe custom union types. An advanced, industrial-strength version is given and explained in detail in [Ale01a]. A discriminated union inspired by the aforementioned article is included in the Boost.Variant library with the name `variant`.

A variant of `variant` is given below, with minor changes but in the same vein.

```
#include <utility>
#include <typeid>
template<typename A, typename B> struct either {
    either(A const& obj): value_one(new A(obj)), value_two(0) {}
    either(B const& obj): value_one(0), value_two(new B(obj)) {}
    either(): value_one(new A), value_two(0) {}
    either(either<A,B> const& obj):
        value_one(obj.is<A>() ? new A(obj.as<A>()) : 0),
        value_two(obj.is<B>() ? new B(obj.as<B>()) : 0) {}
    either<A,B>& operator=(either<A,B> const& obj) {
        *this = (obj.is<A>() ? obj.as<A>() : obj.as<B>() );
    }
    either<A, B>& operator=(A const& obj) {
        this->~either(); value_one=new A(obj);
    }
    either<A, B>& operator=(B const& obj) {
        this->~either(); value_two=new B(obj);
    }
    template<typename T> bool is() const {
        return match(static_cast<T*>(0));
    }
    template<typename T> T& as() {
        if(is<T>()) return *match(static_cast<T*>(0));
        else throw std::bad_cast();
    }
    template<typename T> T const& as() const {
        if(is<T>()) return *match(static_cast<T*>(0));
        else throw std::bad_cast();
    }
private:
    std::auto_ptr<A> value_one;
    std::auto_ptr<B> value_two;
    A* match(A*) { return value_one; }
    B* match(B*) { return value_two; }
};
```

One of the differences between this implementation and the one given in [Ale01a] is that the discriminator (`is`) and accessor (`as`) member functions are distinct, not combined into a `get` function.

## 4.2 Implementation of type conjunction

Type conjunction is tricky to define in C++ because of its limited concept of is-a relations. Since subtyping is equated with substitutability, and subtyping is statically specified, type conjunction has limited applicability in C++. Instead, the extended substitutability outlined in the previous section—namely,

to include implicit convertibility in substitutability—can be. With this point in mind, a simple implementation of type conjunction follows.

```
template<typename A, typename B> struct both {
    template<typename T> both(T const& obj):
        value_one(obj), value_two(obj) {}
    template<typename T> both& operator=(T const& obj) {
        return *this=both(obj);
    }
    operator A() { return value_one; }
    operator B() { return value_two; }
private:
    A value_one;
    B value_two;
};
```

This type accepts all types that can be implicitly converted to both `A` and `B`.

## 4.3 Implementation of the universal type

The universal type, the equivalent of Common Lisp's `t` type, can accept any type whatsoever. Such types are frequently needed in programming, and C and C++ include some built-in facilities that are nevertheless flawed.

The most often used approach, in the absence of a library, is the use of the infamous pointer-to-void type (`void*`). In C, automatic type conversions guarantee that any pointer type can be implicitly converted to `void*`, and vice versa. This guarantee is primarily needed to support C's primary memory allocation mechanism, `malloc`, found in the header `<stdlib.h>`. However, such implicit conversions are dangerous, and may potentially trigger implementation-defined or undefined behavior.

For this reason, C++ outlaws implicit conversions to and from `void*`. Therefore, in C++, the returned pointer value of the `std::malloc` function found within the header `<cstdlib>` must be explicitly cast to the desired pointer type, perhaps with `static_cast`. Thus, using `std::malloc` is (intentially) very verbose and inconvenient; for this reason C++ provides a separate, type-safe, specialized memory allocation mechanism: `new` and `delete`.

With the standardization of the so-called run-time type information (RTTI) mechanism, it is finally possible to write a type-safe, template-based universal type class in standard C++. A thorough examination of this topic is given in [Hen00]; a simplified approach in the same vein follows.

```
#include <utility>
struct any {
    any(): value(new holder<int>(0)) {}
    template<typename T> any(T const& obj):
        value(new holder<T>(obj)) {}
```

```
    any(any const& obj): value(obj.value->clone()) {}
    template<typename T> any& operator=(T const& obj) {
        value=new holder<T>(obj); return *this;
    }
    any& operator=(any const& obj) {
        value=obj.value->clone(); return *this;
    }
    template<typename T> bool is() const {
        return dynamic_cast<holder<T>*>(value);
    }
    template<typename T> T& as() {
        return dynamic_cast<holder<T>&>(*value).value;
    }
    template<typename T> T const& as() const {
        return dynamic_cast<holder<T>&>(*value).value;
    }
private:
    struct holder_base {
        virtual ~holder_base() {}
        virtual holder_base* clone();
    };
    std::auto_ptr<holder_base> value;
    template<typename T> struct holder: holder_base {
        holder(T const& obj): value(obj) {}
        T value;
        holder<T>* clone() { return new holder<T>(value); }
    };
};
```

Note that the `any` class itself is not a template. In order to hold objects of any type in a type-safe manner, templates and object-oriented programming are combined: a type hierarchy of template types (`holder<T>`) with a common base (`holder_base`).

## 4.4 Implementation of the null type

The `null` type of Common Lisp (not to be confused with the `null` value or the `nil` type, which has a sole member—`null`) is the complement of the `t` type: that is, it has no possible values. The C-family programmer will be immediately reminded of the `void` type, which is similar in the sense that no value can be of the `void` type.

However, there are differences. For instance, `void` cannot be in an argument list; a construct such as `f(void)` refers to a function taking *no* arguments, not a function with one argument that is impossible to call. Furthermore, the pointer-to-`void` type in C and C++ is quite different from a literal pointer to void.

Thankfully, a true `null` type is quite easy to define in C++ because of its powerful operator overloading facility:

```
class none { none(); none(none const&); ~none() = 0; };
```

The constructors are all declared private and deliberately undefined in order to prevent instantiation and inheritance, and the destructor is pure virtual, thus preventing any possible destruction.

## 4.5 Implementation of type complement

Type complements, the equivalent of Common Lisp's `not` compound type specifier, accept all values except for values of a certain type. This feature can also be implemented with an approach basically the same as the universal type, with checks added to prevent assignment from certain types.

```
template<typename A> struct except {
    except(): value(new holder<int>(0)) {}
    template<typename T> except(T const& obj):
        value(new holder<T>(obj)) {}
    except(A const&);
    except(except<A> const& obj): value(obj.value->clone()) {}
    template<typename T> except<A>& operator=(T const& obj) {
        value=new holder<T>(obj); return *this;
    }
    except<A>& operator=(except<A> const& obj) {
        value=obj.value->clone(); return *this;
    }
    except<A>& operator=(A const&);
    template<typename T>
    T& as() { return dynamic_cast<holder<T>*>(value)->value; }
    template<typename T>
    T const& as() const {
        return dynamic_cast<holder<T>*>(value)->value;
    }
private:
    struct holder_base {
        virtual ~holder_base() {}
        virtual holder_base* clone();
    };
    std::auto_ptr<holder_base> value;
    template<typename T> struct holder: holder_base {
        holder(T const& obj): value(obj) {}
        T value;
        holder<T>* clone() { return new holder<T>(value); }
    };
};
```

Note how certain functions, such as `except<A>::except(A const&)` and `except<A>::operator=(A const&)`, are deliberately left undefined in order to cause compile-time errors.

## 4.6 Implementation of value-specific types

The Common Lisp `member` and `eql` types are versatile in many cases, when only certain objects within a given type are desired. Using C++'s non-type template parameter mechanism, it can be implemented it as follows:

```
#include <typeid>
template<typename T, T* value1, ..., T* valuen> struct only {
    only(): value(value1) {}
    only(T const& obj) {
        switch(&obj) {
            default: throw std::bad_cast();
            case value1: ... case valuen: value = &obj;
        }
    }
    only(only<T, value1, ..., valuen> const& obj):
        value(new T(*obj.value)) {}
    only& operator=(T const& obj) {
        switch(&obj) {
            default: throw std::bad_cast();
            case value1: ... case valuen: value = &obj;
        }
        return *this;
    }
    only& operator=(only<T, value1, ..., valuen> const& obj) {
        value = obj.value->clone(); return *this;
    }
    operator T() { return *value; }
private:
    std::auto_ptr<T> value;
};
```

Note that the addresses of the possible values must be compile-time constants, and cannot be dynamically allocated. Also, since variadic template parameters are not yet standardized, a template must be defined for each possible number of values (but see §4.8).

## 4.7 Implementation of predicate-specific types

Finally, Common Lisp's `satisfies` compound type specifier provides a set-comprehension-like notation for types. With C++'s function pointers, something similar in C++ can be implemented as follows.

```
#include <typeid>
template<typename T, bool (*p)(T)> struct satisfies {
    satisfies(T const& obj): value(obj) {
        if(!p(obj)) throw std::bad_cast();
    }
    satisfies(satisfies<T, p> const& obj): value(obj.value) {}
    satisfies<T, p>& operator=(T const& obj) {
        if(!p(obj)) throw std::bad_cast();
        value = obj; return *this;
    }
    satisfies<T, p>& operator=(satisfies<T, p> const& obj) {
        value = obj.value; return *this;
    }
    operator T() { return value; }
private:
    T value;
};
```

As with value-specific types, the function pointers must be compile-time constants. Similar templates for member function pointers may be defined as well.

### 4.8 Limitations and future extensions

Even with C++'s powerful template mechanism, C++'s type system is fundamentally static (despite the introduction of RTTI). Imitation of many of Lisp's type facilities is feasible, but they are limited by one common stigma: they must be compile-time constants; that is, the arguments to the class templates must be compile-time constants. This limitation can be partially overcome with pointers.

Also, since C++ currently does not support variadic templates, multiple-value-specific types must be defined for each number of members. Perhaps pointers to arrays or containers can be used, but they are verbose and tedious to use. However, variadic templates are due to be included in the next revision of C++ expected in 2009. [GJMM07] With the introduction of this feature, a number of constructs—including the aforementioned multiple-value-specific types—can be more naturally written.

## 5 Conclusion

I have demonstrated that many powerful type operators found in Common Lisp may be duplicated or imitated in C++ statically using the template mechanism. Even though the static nature of C++'s type system limits the type operators' usefulness, they are, nevertheless, a useful and expressive extension of the C++ type system and a testament to C++'s immense power.

# References


[AG04] David Abrahams and Alekesy Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. C++ In-Depth Series. Addison-Wesley Professional, December 2004.

[Ale01a] Andrei Alexandrescu. An implementation of discriminated unions in C++. In *C++ Template Workshop*, 2001.

[Ale01b] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. C++ In-Depth Series. Addison-Wesley Professional, February 2001.

[GJMM07] Douglas P. Gregor, Jaako Järvi, Jens Maurer, and Jason Merrill. Proposed wording for variadic templates. Document N2152=07-0012, ISO/IEC JTC1 SC22 WG21, January 2007.

[Gra93] Paul Graham. *On Lisp: Advanced Techniques for Common Lisp*. Prentice Hall, September 1993.

[GS06] Douglas P. Gregor and Bjarne Stroustrup. Concepts (revision 1). Document N2081=06-0151, ISO/IEC JTC1 SC22 WG21, September 2006.

[Hen00] Kevlin Henney. Valued conversions. *C++ Report*, 12(7), July–August 2000.

[INC94] International Committee for Information Technology Standards. *INCITS 226:1994 [R2004] Programming Language Common Lisp*, January 1994.

[Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003.

[Koe03] Andrew R. Koenig, editor. *ISO/IEC 14882:2003 Programming Languages—C++*. International Organization for Standardization, October 2003.

[KS05] Sunil Kothari and Martin Sulzmann. C++ templates/traits versus Haskell type classes. Technical Report TRB2/05, The National University of Singapore, October 2005.

[LDG+05] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml System Release 3.09: Documentation and User's Manual*. Institut National de Recherche en Informatique et en Automatique, October 2005.

[Mey95] Scott Douglas Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Professional Computing Series. Addison-Wesley Professional, December 1995.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. MIT Press, revised edition, May 1997.

[RS05] Gabriel Dos Reis and Bjarne Stroustrup. A formalism for C++. Document N1885=05-0145, ISO/IEC JTC1 SC22 WG21, October 2005.

[SA04] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. C++ In-Depth Series. Addison-Wesley Professional, October 2004.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 3rd edition, June 1997.

[Vel03] Todd L. Veldhuizen. C++ templates are Turing complete, June 2003. Available at `http://osl.iu.edu/~tveldhui/papers/2003/turing.pdf`.