졸업 논문

Implementation of Common Lisp-Like Types Using Generic Metaprogramming

한국과학영재학교

김 형 록

Contents

Implementation of Common Lisp-like Types Using Generic Programming

Hyungrok Kim
Korea Science Academy

March 16, 2007

Abstract

The Common Lisp Type System is very rich and offers many features rarely found in more statically typed languages. In part, this is because such features have been difficult to implement in the past. However, with the introduction of generic programming techniques, it is now possible to emulate various aspects of the Common Lisp Type System.

This paper explores how Common Lisp's compound type specifiers may be viewed as metafunctions and can consequently be implemented in generic programming features commonly found in major programming languages, such as C++. Features needed for effective implementation and use of such metafunctions are analyzed, and current languages' support for these features is evaluated.

Keywords: Common Lisp, generic programming, templates, C++, D

1 Introduction

Traditionally, programming languages have been divided into two categories, depending on their typing disciplines: statically typed languages, such as Ada, C, C++, D, Eiffel, Fortran, Haskell, Java, ML, Modula, and Pascal, have a built-in static type system designed to ensure that type-safety violations do not occur, while in dynamically typed programming languages, such as Lisp, Smalltalk, Javascript, Python, and Ruby, most type-checking is done in execution-time. Thus, dynamically typed languages offer more freedom in type checking. Because of the built-in nature of the type systems of statically typed languages, 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).

One of the features found in more recent statically typed languages designed to overcome this restriction is generics or parametrized types. Although the full powers of generics were not realized when it was first designed, it is now known today that powerful generics systems can offer full, Turing-complete computation at compile time.[23] In particular, a form of compile-time programming, called *template metaprogramming* in C++, can be used to extend the type system in many ways. For instance,[4] (Ch. 11) explains how multimethods may be implemented with template metaprogramming.

One of the more distinctive features of Common Lisp is its expressive, flexible type declaration system, with various operators for type algebra. This paper considers how Common Lisp's type system can be implemented in statically typed languages using generic programming features.

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[19] 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[19] (§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

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.

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 include 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 the type (and (satisfies oddp) (satisfies perfectp)) (where oddp and perfectp denote predicates for odd numbers and perfect numbers, respectively) is a subset of nil . 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.[19] (§4.4)

It is also possible to create user-defined compound type specifiers (*derived type specifiers* in the terminology of the standard) with a macro called deftype.[19] (§4.4) Derived type specifiers are lexically scoped and may be an arbitrary Lisp form that, when evaluated, results in a predefined compound type specifier.

3 Metafunctions

C++ templates, and generic programming features in earlier languages such as Ada 89, were not intended as tools for metaprogramming. The use of templates as a tool for metaprogramming was first realized in 1995[22], more than five years after the introduction of templates to the C++ language. Since then, template metaprogramming has been used in many applications to augment and extend the compile-time features of the language.

The seminal book[2] formally defines the concept of metafunctions in the context of C++: according to the book,

A *metafunction* is either

- a class template, all of whose parameters are types

or

- a class

with a publicly accessible nested result type called type.

In order to include metafunctions returning values (which the authors call 'numerical metafunctions'), the book notes that the type can be "a type known as an integral constant wrapper, whose nested ::value member is an integral constant." The authors go on to describe the Boost Metaprogramming Library[1], a C++ library which formalizes the concept of a metafunction concretely and includes utilities to manipulate them.

However, it should be evident that the above definition is C++-specific and overly concrete. In this paper, a more conceptual and general definition will be used: a *metafunction* is a function-like entity which takes zero or more parameters, which may be types or values (compile-time or otherwise), and which returns a type or a value. It might be evaluated compile-time (as in Lisp macros and C++ templates) or run-time. Note that this definition includes normal functions as a subset. Also note that this definition is more general than[2] in that metafunctions need not be compile-time only. One should also note that in a dynamic language where types are run-time values, every metafunction becomes, in effect, a plain function.

An example of a metafunction would be a maximumValue metafunction that takes a numeric type as a parameter and returns its maximum value. For instance, maximumValue(UnsignedInteger32) might return $2^{32} - 1$, while maximumValue(FloatingPoint) might return 10^{37}. Note that this is similar to C++'s std::numeric_limits<T>::max() class template member function; in fact, std::numeric_limits<T> is an example of the C++ technique called *traits*, which is one of the examples of metafunctions given in[2].

Why use metafunctions?[2] primarily focuses on two areas: compile-time numerical computations and embedding domain-specific language (DSLs) within a host language. Metafunctions as used in the former area are, in a sense, not "proper" metafunctions in that they neither accept nor return a type: they may be viewed as plain functions that happen to be evaluated compile-time. In fact, languages such as Lisp and D have constructs that direct the compiler to evaluate an expression at compile-time, making this use of templates needless.

It is the other use that is of interest in this paper. In order to define a domain-specific language, the type system of the host language must be extended in various directions as to harmonize the DSL with the host language.[2] demonstrates that this can be done very successfully and efficiently with C++ templates and similar generic programming features found in other languages.

4 Compound Type Specifiers as Metafunctions

It should now be obvious that Common Lisp's compound type specifiers are instances of metafunctions returning types. In the case of Lisp, this is especially obvious: the syntax of complex type specifiers, like most Lisp constructs, is identical to that of a function call or a macro call—parentheses. In addition, the evaluation of user-defined types with deftype is essentially indistinguishable from that of functions or macros.

It should also be evident that, as type-returning metafunctions, Lisp's compound type specifiers can be implemented in generic programming features in statically typed languages. So, for instance, the type (or A B) in Lisp might be written as Or<A, B> in C++ and the type (satisfies A pred) in Lisp might be written as Satisfies<A, &pred> in C++.

One can question whether a facility only found in a language as dynamic and flexible as Common Lisp has any value in a statically typed language. However, it should be noted that many ad-hoc emulations of certain elements of the Common Lisp type system (probably without awareness of preexistent facilities in Lisp) have been present in statically typed languages.

For instance, variant records in languages such as Ada, Pascal, Modula-2, Algol 68 (where they are called "united modes"), and C and C++ (where they are called unions) may be viewed as a built-in emulation of Common Lisp's or complex type specifier.

However, the C/C++ union construct is untagged and may cause undefined, dangerous, unpredictable behavior when used without strict discipline.[21] (§97) In order to resolve this problem, a number of template-based techniques have been proposed that alleviate the type safety problem. Perhaps the most advanced of these is Alexandrescu's implementation, given in[3],[5],[6], and[7], and the Boost.Variant library[12] inspired by Alexandrescu's design. These template-based libraries can be viewed as essentially a template metafunction implementation of Common Lisp's or compound type specifier.

Another interesting example of preexisting emulations of Lisp's type system is the so-called variant type found in Visual Basic versions prior to the release of Visual Basic .NET. When a variable is declared without an explicitly stated type in classic Visual Basic, e.g. Dim A (or explicitly declared as a variant type, e.g. Dim A As Variant), it can take any value—string, integer, floating-point number, etc. Although this feature is considered dangerous and was removed in Visual Basic .NET, it is still indispensable in certain situations. For instance,[20] (Item 36) gives the motivating example of implementing a weakly typed scripting language in which variables can hold any type.

Many techniques are used to simulate this in languages that do not have variant types built-in, such as C and C++;[17] mentions that "[t]he common solution [in C and C++] has been to use a union or a void pointer". However, such techniques are fragile and unsafe and may result in undefined, unpredictable behavior. In order to resolve this problem, a number of template-based solutions have been proposed, such as[15] and the Boost.Any library[16] based on it,[10], and[11]. This is essentially Common Lisp's t type, the universal ancestor of all

types. The Object types of some object-oriented languages, such as SmallTalk, Eiffel, and C# (and, to a lesser degree, Java, due to its "primitive" types), serve a similar function.

Still another example is the Singleton design pattern, first popularized by the seminal book *Design Patterns*[13]. In it, a technique based on visibility and encapsulation features is used to prevent multiple instantiations of a type. However, this technique is repetitive and cannot be easily formalized. Also, if the language lacks visibility-related features (as is the case in Python), the Singleton design pattern cannot be easily used. Languages such as Ruby have singleton classes built into the language. An easier means may be to use a type which, by definition, has a single element, such as that provided by Common Lisp's eql type specifier. For instance, in the Common Lisp Object System, an eql specializer can be used to attach a method to a specific object, e.g. (defmethod method ((arg (eql singleton))) ...).

Finally, enumeration types are types that can only have a fixed set of values, found in languages such as C++ (but not C, as enum constructs are not distinct from integers), Java, C#, and Ada. In languages without it, such as Java prior to the release of J2SE 5.0, complicated, informal design patterns using visibility features (such as the Typesafe Enum pattern in[8] (Item 21)) need to be used. Furthermore, in less weakly typed languages, such as C++, integers may be cast to enumerations, which may not be desirable. It should be clear that enumeration types are equivalent to Common Lisp's member compound type specifier, which constructs a type from a list of possible values.

However, there are still features from the Common Lisp type system that are not built-in or commonly emulated in statically typed languages but are nevertheless useful and convenient. An example is the satisfies compound type specifier, which is the direct analogue of the set-builder notation, e.g. (satisfies A p) is equivalent to $\{x \in A | p(x)\}$ in mathematics. This can be used to place constraints in arguments of functions. For example, a function that reads a character from a file might be declared as Char readChar(Satisfies<File, both(isOpen, complement(isEof))> file), which states that the argument file must be open and must not be at the end of the file. It should be noted that this is similar to the Design by Contract feature[18], found in languages such as Eiffel and D.

5 Implementation Issues

In order to allow a full and well-integrated implementation of Lisplike types with templates, a number of language features are required, some of which are not found in some languages. The presence of these features in common programming languages with generic programming features is summarized in Table 2. These features will each be explained in detail in subsequent sections.

Language	Template Parameter Requirements	Overloading Assignment	User-Defined Subtyping	Variadic Templates
Ada	Types, objects, subprograms (procedures and functions), and packages	Impossible	Specialized subtype syntax for numbers, enumerations, records, arrays, etc.; arbitrary subtyping not possible, not even inheritance	Impossible
C++	Types, pointers (to objects, to functions, and to members), references, integral constants, templates, but not floating-point constants or string literals	Possible	Inheritance only; user-defined casts can compensate, but are inadequate	Impossible; planned to be included in C++0x[14]
C#	Types only	Impossible	Inheritance only; user-defined implicit cast operators can compensate	Impossible
D	Types, pointers (to objects and to functions), templates, integral constants, arbitrary names, floating-point constants, string literals, modules	Possible	Inheritance only; user-defined casts may not be dependent on target type	Possible
Eiffel	Types only	Impossible	Inheritance only	Impossible
Java	Classes only; primitive types not allowed	Impossible	Inheritance only	Impossible

Table 2: Comparison matrix for features needed for the implementation of Lisplike metafunctional types.

5.1 Restrictions on generic parameters

In many languages, only types can be passed to generic types. However, this is problematic in the implementation of some compound type specifiers, such as satisfies, that takes a predicate or other function. Also, compound type specifiers like eql and member require values to be passed to the generic type. In such languages, these kinds of compound type specifiers cannot be implemented easily.

Another problem arises from the compile-time nature of the generic programming features of some languages. In order to remove any run-time inefficiency, these languages sometimes require generic parameters to be compile-time constants. In some cases, these problems may be circumvented by passing the address of the object in question instead, but require extra syntax.

5.2 User-defined subtyping

In many instances, it may be desirable to designate a compound type specifier to be the subtype of another type. For instance, if one were to implement a Satisfies<A,pred> type, one would like it to be a subtype of A. However, in most object-oriented languages, inheritance is the only means of achieving subtyping (a notable exception is Smalltalk). This places a heavy implementation restriction, since some classes may not be designed for inheritance (indeed, many languages have features or techniques to prevent inheritance from a class), and in any case inheritance slows down execution by method dispatching.

Another problem is that the type hierarchy frequently needs to be a general directed acyclic graph. For instance, we would like Eql<a> to be the subtype of all of Member<a, b>, Member<a, b, c>, etc. However, if we equate subtyping with inheritance, this would imply that multiple inheritance is needed. However, many language lack multiple inheritance, and no language can support implementation inheritance from an infinite list of parents.

These problems can be alleviated to a degree with user-defined implicit type casts. However, some languages, such as Java and Eiffel, do not support user-defined type casts on the ground that they are dangerous and may be misused. In other languages, such as C++, the rules for applying user-defined casts may be overly elaborate and prone to confusion. A third problem is that in some languages, such as D, the type cast operator may not know to which type it is being cast, which renders it useless if the type is to be converted to many other types.

5.3 Variadic generics

When implementing some compound type operators, such as or, it is frequently desirable to be able to generalize it to an arbitrary number of parameters, i.e. to be able to write Or<A, B, C, D> rather than Or<A, Or<B, Or<C, D>>>. In order to be able to implement such types, a language feature called *variadic generics* is needed. Variadic generics are generics that can take an arbitrary

number of arguments. However, few languages support variadic generics; the D programming language[9] supports it, and the upcoming C++0x is planned to support it[14].

6 Conclusion

I hope the reader has realized that generic programming is an effective tool to extend the type systems of statically typed languages and the importance of many of the type facilities that are offered in Common Lisp and now in more statically typed programming languages as well, thanks to generic programming. Using such features, previously informally described design patterns can be formalized and modularized, and many previously rarely seen techniques can be utilized with ease, enabling clearer and more effective programming.

References

[1] D. Abrahams and A. Gurtovoy. The Boost C++ metaprogramming library. Available on the Internet at `http://boost.org/libs/mpl/doc/paper/mpl_paper.pdf`, Mar. 2002.

[2] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. C++ In-Depth Series. Addison-Wesley Professional, Dec. 2004.

[3] A. Alexandrescu. An implementation of discriminated unions in C++. In *Proceedings of OOPSLA 2001: Second Workshop on C++ Template Programming*, Oct. 2001. Available on the Internet at `http://oonumerics.org/tmpw01/alexandrescu.pdf`.

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

[5] A. Alexandrescu. Discriminated unions (I). *C/C++ Users Journal*, 20(4), Apr. 2002.

[6] A. Alexandrescu. Discriminated unions (II). *C/C++ Users Journal*, 20(6), June 2002.

[7] A. Alexandrescu. Discriminated unions (III). *C/C++ Users Journal*, 20(8), Aug. 2002.

[8] J. Bloch. *Effective Java Programming Language Guide*. The Java Series. Prentice Hall PTR, June 2001.

[9] W. Bright. The D programming language. *Dr. Dobb's Journal*, 27(2):36–40, Feb. 2002.

[10] F. L. Cacciola Carballal. An improved variant type based on member templates. *C/C++ Users Journal*, 18(10):10–21, 2000.

[11] C. Diggins. An efficient variant type. *C/C++ Users Journal*, 23(11), 2005.

[12] E. Friedman and I. Maman. The Boost.Variant library documentation. Available on the Internet at `http://boost.org/libs/variant`, Mar. 2003.

[13] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley Professional, Jan. 1995.

[14] D. P. Gregor, J. Järvi, J. Maurer, and J. Merrill. Proposed wording for variadic templates. Document N2152=07-0012, ISO/IEC JTC1 SC22 WG21, Jan. 2007.

[15] K. Henney. Valued conversions. *C++ Report*, 12(7), July–Aug. 2000.

[16] K. Henney. The Boost.Any library documentation. Available on the Internet at `http://boost.org/libs/any`, Nov. 2005.

[17] J. Hyslop and H. Sutter. I'd hold anything for you. *C/C++ Users Journal*, 19(12), Dec. 2001.

[18] B. Meyer. Design by contract. Technical Report TR-EI-12/CO, Interactive Software Engineering, Inc., 1986.

[19] K. M. Pitman, K. Chapman, et al. *INCITS 226:1994 [R2004] Programming Language Common Lisp*. International Committee for Information Technology Standards, Jan. 1994.

[20] H. Sutter. *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions*. C++ In-Depth Series. Addison-Wesley Professional, Aug. 2004.

[21] H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. C++ In-Depth Series. Addison-Wesley Professional, Oct. 2004.

[22] T. L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

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

A Appendix: Implementation Examples

The subsequent examples are in a pseudocode-like imaginary language with ideal support for all the features discussed in Section 5. They may or may not be able to be translated to a particular language if the target language does not support some of the necessary features. Constructs like static if and static for are executed in compile time, like D's static if. Template specialization is used. The cast method is invoked for implicit type casts.

A.1 Implementation of Or

This is a customary tagged union type, the equivalent of Common Lisp's or compound type specifier.

```
1  // Whether the first argument is contained in the rest
2  private template<type A>
3    const boolean contains := false;
4  private template<type A, A, type B...>
5    const boolean contains := true;
6  private template<type A, type B, type C...>
7    const boolean contains := contains<A, C...>;
8
9  // whether the first set of types is contained in the second
10 private template<<type A, type B...>, <type C...>>
11   const boolean subset := contains<A, C...> && subset<<B...>, <C...>>;
12 private template<<>, <type C...>>
13   const boolean subset := true;
14
15 public template<type A...> class Or isa A... {
16   static foreach(type T: A...) {
17     private A value<A> := null;
18   }
19
20   public template<type T> void constructor(T value) {
21     value<T> := value;
22   }
23
24   public template<type T> T cast() throws CastError<T> {
25     if(value<T> == null) {
26       throw new CastError<T>();
27     } else {
28       return value<T>;
29     }
30   }
31
32   public template<type T> void :=(T newValue) {
```

```
    static foreach(type T: A...) {
      value<T> := null;
    }

    value<T> := newValue;
  }

  public template<type T...> void :=(Or<T...> newValue) {
    static if(subset<<T...>, <A...>>) {
      static foreach(type B: A...) {
        static if(contains<B, T...>) {
          value<B> := newValue<B>;
        } else {
          value<B> := null;
        }
      }
    }
  }

  public template<type T> boolean is() {
    return value<T> != null;
  }

  public template<type T> T get() throws CastError<T> {
    if(is<T>()) {
      return value<T>;
    } else {
      throw new CastError<T>();
    }
  }
}
```

A.2 Implementation of Satisfies

This is an implementation of the equivalent of Common Lisp's or compound type specifier.

```
public template<type T, bool function(T) predicate> class Satisfies isa T {
  private T value;
  public T constructor(T initialValue) throws InvalidArgumentError {
    if(predicate(value)) {
      value := initialValue;
    } else {
      throw new InvalidArgumentError(initialValue);
    }
  }
```

```
10
11    public void :=(T newValue) throws InvalidArgumentError {
12      if(predicate(newValue)) {
13        value := newValue;
14      } else {
15        throw new InvalidArgumentError(newValue);
16      }
17    }
18
19    public T cast() {
20      return value;
21    }
22  }
```