

An Implementation of Discriminated Unions in C++

Andrei Alexandrescu
andrei@metalanguage.com

Synopsis

Discriminated unions (also known as variant types or tagged unions) are data structures that hold one object of any of a set of types, together with means to identify the actual type stored. Discriminated unions are useful in applications such as interpreters, database programs, and data communications. A number of implementations in C++ have been published [1], [2]. This paper describes an implementation of generic discriminated unions in C++ that combines the following features: (1) the ability to specify the set of possible types accepted, (2) transparency among built-in and user-defined types, (3) “total decoherence” mode in which the user is required to treat all possible types of a discriminated union, and get a compile-time error otherwise, and (4) high efficiency by avoiding free store usage and by offering O(1) conversions. The features (1) and (3) are novel in comparison to other implementations.

This paper uses and assumes knowledge of the Loki C++ library [3]. Loki offers generic components for design patterns and idioms so its user doesn’t have to build designs starting from first principles. The components used are typelists, Visitor, and hierarchy generators.

1. Introduction

Many applications need to store unrelated data types in a uniform format. For example, consider an application that communicates with a relational database. The database can store fields of types such as strings, integers, decimal, floating point, and date. When querying the database, the results come in the form of tables that can contain any of these types as columns. When writing a querying function, the C++ program must adapt these types to a uniform format so it can store the results of any query uniformly (in a bi-dimensional array, for example).

Current C-heritage database APIs store the field value as a union of all possible types, together with an integral or enumerated value (a “tag”) that specifies which field of the union is valid. For example:

```
// Example 1: a C-style approach to representing
// a database field type
struct DatabaseField
{
    enum TypeTag { typeNull, typeInt, typeDouble,
                  typeString, typeDate }
    tag_;

    union
    {
        int fieldInt_;
        double fieldDouble_;
        char* fieldString_;
        Date typeDate_;
    };
    ...
};
```

Needless to say, this approach is clumsy and error-prone.

Object-oriented approaches to interfacing with relational databases prescribe using polymorphic objects for fields, approach having the advantage of type safety. However, this advantage is dwarfed by the awkwardness of the design. Object-oriented programming (at least when it is statically typed as in C++) prescribes defining a uniform interface for types in a hierarchy, which is not realizable in this case because heterogeneous types such as string, integer, date, or Boolean have no sensible common interface. Ultimately this approach leads to either abundant casts or a bloated, error-prone interface, which essentially defines a runtime-checked getter for each type.

As another example for the need for discriminated unions, dynamically typed languages (LISP, Basic) use variables of which type changes depending on the value assigned. Upon access, the language

infers and checks the type depending on context. Implementing such types poses challenges similar to those of database programming mentioned above.

2. Type-safe Discriminated Unions

Example 1 above describes a C-style design of a discriminated union. To eliminate the disadvantages of that approach, the following steps must be taken:

- *Introduce type safety.* The tagged union approach does not offer any type safety. A mechanism that is type-safe without degrading performance must be introduced.
- *Generalize the collection of possible types.* For discriminated unions to be general, the set of accommodated types must not to be hardcoded in a `union`, but rather user-configurable from the outside.
- *Support user-defined types in addition to primitive types.* A glossed-over detail in Example 1 above is the ownership of the `fieldString_` member. Usually, `DatabaseField`'s destructor `delete[]`s the `fieldString_` member if `tag_` equals `typeString`, artifact that opposes generalization. The most desirable approach is to store the string field in an elaborate user-defined type, such as `std::string`. However, a C++ `union` cannot store a type with constructors or a destructor. A generic approach to discriminated unions should support built-in and user-defined types seamlessly.

A possible approach to generalizing the set of types supported by a discriminated union type is to use typelists. Typelists [described in chronological order in 7, 8, 9, 3] allow manipulating lists of types, offering similar functionality to lists of values. The rest of this paper makes use of the `Loki::Typelist` class template as defined in [3].

The desired synopsis of the to-be-defined `Variant` class therefore is:

```
template <class TList>
class Variant
{
    ...
};
```

The user can define a `Variant` containing a specific set of types, by instantiating `Variant` with a `Typelist` instantiation:

```
// Example 2: A typelist-driven approach to discriminated unions
```

```
typedef Variant<
    TYPelist_4(int, double, std::string, Date)>
    DatabaseField;
```

At a minimum, the primitives of the `Variant` type must include:

- Value semantics: default constructor, copy constructor, assignment operator, and non-leaking destructor;
- Constructors that accept any type contained in the passed-in typelist;
- Assignment operators that accept any type contained in the passed-in typelist;
- A type-safe means to convert to any of the types in the passed-in typelist;
- A function that returns the actual type currently stored by the `Variant`;
- Support for efficiently swapping two `Variant` objects, for using in certain idioms and in some STL algorithms;
- A means to efficiently change the type of a `Variant` object in place if a sensible conversion exists, for example, changing its type from `int` to `double`. This is needed for increasing flexibility and uniformity in `Variant` manipulation.

3. Storage

The consecrated approach to implementing flexible type-safe discriminated unions in C++ (see [2]) is to combine a polymorphic approach with the “pimpl” idiom [4]. This design can implement all the features described above and more, at the expense of unnecessarily using the free store for allocation.

The implementation described in this paper stores any of the types right in the `Variant` object. To do this, `Variant` needs a helper compile-time algorithm that computes the maximum size of a type in a typelist, algorithm depicted below.

```

template <class TList> struct MaxSize;
template <class T>
struct MaxSize<Loki::NullType>
{
    enum { result = 0 };
};
template <class Head, class Tail>
struct MaxSize<Loki::Typelist<Head, Tail> >
{
private:
    enum { tailResult = size_t(MaxSize<Tail>::result) };
public:
    enum { result = sizeof(Head) > tailResult ?
           sizeof(Head) : size_t(tailResult) };
};

```

For any typelist `TList`, `MaxSize<TList>::result` returns at compile time the maximum size of all types contained in `TList`. Using `MaxSize`, `Variant` stores its data in the following manner (alignment issues will be dealt with in the next section):

```

template <class TList>
class Variant
{
    enum { size = MaxSize<TList>::result };
    unsigned char buffer_[size];
    ...
};

```

In addition, `Variant` must store the discriminator, a tag that helps identifying the correct type stored in the raw buffer. A simple approach would be to use an integral tag; for flexibility and speed reasons, `Variant` stores a pointer to a static array of pointers to functions – an emulated virtual table. Section 5 describes the type tag implementation.

4. Computing Alignment

No approach to computing alignment in C++ is entirely portable, because the language lacks the appropriate primitives (such as the `__alignof__` keyword implemented by some compilers as an extension). However, with some effort and a number of reasonable assumptions, alignment can be computed accurately on a large set of platforms.

The alignment problem to solve is: given a typelist, return a POD (plain old data) type that guarantees proper alignment for any type in the typelist.

For computing alignment, let's first consider a typelist `TypesOfAllAlignments` that contains types of various alignments:

- a) All primitive types
- b) Pointers to all primitive types
- c) A pointer to function
- d) A pointer to member variable
- e) A pointer to member function
- f) A class having a virtual function

For each type mentioned in (a) – (e), add a POD structure containing exactly one member of that type. This addition is necessary because some compilers align structures differently than primitive types, even when they are structurally equivalent to primitive types.

The `AlignmentCalculator` algorithm that computes alignment for the types in a typelist `TList` is the following:

- Assign `TypesOfAllAlignments` to `Temp`.
- Compute the maximum size of all types contained in `TList` using the `MaxSize` class template described in Section 3 above. Store the result in the compile-time constant `maxSize`.
- Remove from `Temp` all types that have a size greater than `maxSize`.

The result is a union type containing a member of each type in `Temp`. Constructing the union is possible because `Temp` contains POD types only. (Building the union directly from `TList` is not possible because `TList` might contain user-defined types with constructors or destructors.)

The common approach to ensuring proper alignment is simply to define a union containing every type in `TypesOfAllAlignments`. That type is likely to have the maximum alignment requirements, at the expense of occupying extra memory. An interesting property of the `AlignmentCalculator` algorithm is that, unlike the simpler approach, `AlignmentCalculator` computes the alignment without incurring size overhead and without sacrificing portability.

Having `AlignmentCalculator` in place, `Variant` enforces alignment in the following manner:

```
template <class TList,
          typename Align = AlignmentCalculator<TList>::Result>
class Variant
{
    enum { size = MaxSize<TList>::result };
    union
    {
        unsigned char buffer_[size];
        Align dummy_;
    };
    ...
};
```

The alignment is specified as a template parameter and defaults to the computed value. This enables `Variant` users on platforms with unusual alignment requirements to specify an alignment type if necessary, without changing `Variant`'s code.

The storage structure defined above enables `Variant` to store any primitive or user-defined type right inside its body, without performing any free store allocation.

5. The Type Tag

As briefly mentioned, `Variant` performs type identification and manipulation through a pointer to a table of functions. This is a generic, efficient approach. The structure of the table is depicted below.

```
template <...>
class Variant
{
    struct vTable
    {
        const std::type_info& (*typeId_ )();
        void (*destroy_)(const Variant&);
        void (*clone_)(const Variant&, Variant&);
        void (*cloneTypeOnly_)(const Variant&, Variant&);
        void (*swap_)(void* lhs, void* rhs);
        bool (*changeType_[Loki::TL::Length<TList>::value])
            (Variant&);
        ...
    };
    vTable* vptr_;
    ...
};
```

The identifier names used above hint to the similarity between the structure used and the so-called “vtable” and “vptr” – jargon established by the classic implementation of virtual functions in C++. The functions’ roles are described below:

- `typeId_` points to a function that returns the corresponding `std::type_info` for the object currently stored.
- `destroy_` is invoked during destruction of the `Variant` object.
- `clone_` points to a function that duplicates a `Variant` object.
- `cloneTypeOnly_` points to a function that creates an empty clone of a `Variant` object (clones the type only, but doesn’t copy the value).
- `swap_` swaps the content of two `Variant` objects.
- `changeType_` is a fixed-size array of functions that change the type of the `Variant` object to any other type specified in the typelist with which the `Variant` is instantiated.

VTable needs pointers to functions for initializing its members. These are provided by the VTableImpl class template, also embedded inside Variant:

```
template <...>
class Variant
{
    template <class T>
    struct VTableImpl
    {
        static const std::type_info& TypeId()
        {
            return typeid(T);
        }
        static void Destroy(const Variant& var)
        {
            const T& data =
                *reinterpret_cast<const T*>(&var.buffer_[0]);
            data.~T();
        }
        ...
        static VTable vTbl_;
    };
    ...
};
```

For any type T, VTableImpl<T> has static member functions corresponding to all VTable's members, and holds a static VTable member variable, called vTbl_. When initializing a Variant object, its vptr_ is initialized to point to the appropriate VTable<T>::vTbl_ depending on the type T with which the Variant is initialized. The STATIC_CHECK macro, defined by Loki, checks a compile-time Boolean constant and engenders a compile-time error if the constant is false.

```
template <...>
class Variant
{
    ...
public:
    template <class T>
    Variant(const T& val)
    {
        STATIC_CHECK((Loki::TL::IndexOf<TList, T>::value >= 0),
            Invalid_Type_Used_As_Initializer);

        new(&buffer_[0]) T(val);
        vptr_ = &VTableImpl<T>::vTbl_;
    }
};
```

After ensuring that the passed-in type belongs to the supported set of types, a T is constructed in buffer_'s space and then the vptr_ member is set to point to a VTableImpl instantiation that's specialized in handling objects of type T.

Because buffer_ and vptr_ are initialized in coordination, the functions in VTableImpl can safely assume that buffer_ contains a T object and use reinterpret_cast to access it.

Once vptr_ is properly initialized, Variant can implement all of its functionality by forwarding to the pointers to functions stored in the vTbl_.

Let's focus on the initialization of VTableImpl<T>::vTbl_. Each VTableImpl<T> must pass the type T to a constructor of VTable. This is doable with Loki::Type2Type simple template, which serves to carry type information about a type without the overhead of creating a value of that type.

VTable's constructor, in turn, is templated in T and accepts a dummy argument of type Loki::Type2Type<T>. It then proceeds to initialize each pointer to function with addresses of the functions implemented in VTableImpl:

```
template <...>
class Variant
{
    ...
    struct VTable
    {
        template <class T>
```

```

vTable(Loki::Type2Type<T> tt)
{
    typeId_ = &VTableImpl<T>::TypeId;
    destroy_ = &VTableImpl<T>::Destroy;
    clone_ = &VTableImpl<T>::Clone;
    cloneTypeOnly_ = &VTableImpl<T>::CloneTypeOnly;
    swap_ = &VTableImpl<T>::Swap;

    Init(changeType_, tt, TList()); // see below
}
...
};
...
};

```

6. Conversions

A member of `vTable` that deserves special attention is:

```
bool (*changeType_[Loki::TL::Length<TList>::value])(Variant&);
```

The type of `changeType_` is array (of length `Loki::TL::Length<TList>::value`) of pointers to functions taking a `Variant&` and returning `bool`. This array serves for changing the type of a `Variant` object in-place. The semantics of the N^{th} function in this array is: if a conversion from the current type to the N^{th} type in the typelist exists, the function converts the `Variant` object to that N^{th} type and returns `true`. Otherwise, the function returns `false`.

There is one array slot per type in the typelist of supported types. Therefore, converting from any type to any other type is $O(1)$ with respect to the number of types stored.

Ideally, the `changeType_` member, as all other members of the static `vTable` objects, would be initialized at compile time. This is not possible because in C++ the static array initialization syntax doesn't lend itself to a compile-time expansion of generic code.

Detection of convertibility is made by using the `Loki::Conversion` class template, described in [3]. `Conversion` makes possible to select at compile time between a no-op function that returns `false`, and a function that performs the cast and returns `true`.

Below is the code that initializes the `changeType_` array.

```

template <...>
class Variant
{
    ...
    struct vTable
    {
        ...
        template <class T, class TList>
        void Init(bool (**pChangeType)(Variant&),
            Loki::Type2Type<T> tt, TList)
        {
            typedef typename TList::Head Head;
            typedef typename TList::Tail Tail;
            enum { canConvert =
                Loki::Conversion<Head, T>::exists != 0 };
            *pChangeType = &VTableImpl<T>::
                Converter<Head, canConvert>::Convert;
            Init(pChangeType + 1, tt, Tail());
        }

        template <class T>
        void Init(bool (**)(Variant&), Loki::Type2Type<T>,
            Loki::NullType)
        {
            // nothing to do - stop recursion
        }
    };
};

```

The `Init` function template is recursive at compile time. Each call to `Init` initializes one function pointer, and performs a tail recursion to initialize the rest of the pointers. The recursion is realized by `Init`'s last parameter, which is a typelist that `Init` reduces (beheads) with each recursive call. Finally, the

overload of `Init` that accepts a `Loki::NullType` object (the terminator of any typelist) stops the recursion.

`Converter` is a simple class template specialized for two cases – conversion exists or doesn't exist, and exposes one static function `Convert` that realizes the conversion.

7. Decoherence and Visitability

In the context of this paper, “decoherence” means finding the actual type stored in a `Variant` object. The term is borrowed from quantum mechanics, where “decoherence” stands for switching a quantum system from quantum-specific, counterintuitive, behavior, to classical behavior. Similarly, for a `Variant` object, “decoherence” means finding the “classic” C++ type (and object) that the `Variant` cloaks.

The simplest decoherence mechanism that `Variant` implements comes in the form of a templated `GetPtr` function. `Variant<TList>::GetPtr<int>()` returns a pointer to the stored object if it is of type `int`, and zero otherwise.

```
template <...>
class Variant
{
public:
    ...
    const std::type_info& TypeId() const
    {
        return (vptr_>typeid_());
    }
    template <typename T> T* GetPtr()
    {
        return TypeId() == typeid(T)
            ? reinterpret_cast<T*>(&buffer_[0])
            : 0;
    }
};
```

A `const` version exists for `GetPtr`. Similarly, two convenience function templates `Get` are added. They return a reference instead of a pointer, and throw `std::runtime_error` if the type asked is not the same as the type stored in the `Variant`. As a side note, if you store `T` and try to extract a `const T` from a non-`const Variant`, the attempt will fail. This behavior is by design.

Using `GetPtr` and `Get`, it is possible to access the object held by a `Variant` object, provided you know its type:

```
typedef Variant<
    TYPELIST_4(int, double, std::string, Date)>
    DatabaseField;
...
DatabaseField fld;
if (int* pInt = fld.GetPtr<int>())
{
    ... pInt points to the integer stored ...
}
else if (double* pDb1 = fld.GetPtr<double>())
{
    ... pDb1 points to the double stored ...
}
else if (std::string* pStr = fld.GetPtr<std::string>())
{
    ... pStr points to the string stored ...
}
else if (Date* pDate = fld.GetPtr<Date>())
{
    ... pDate points to the Date stored ...
}
else
{
    assert(false); // can't reach here
}
```

The problem with this hand-made decoherence is that the code is brittle in the face of change. If the user later adds to the collection of types that the `Variant` holds, the compiler won't ensure that all types are checked by the burst of `if/else` statements (the `assert` above will fire).

A better compiler-checked decoherence mechanism is applying the Visitor pattern [5] to `Variant`. There is a strong correlation between discriminated unions and the Visitor pattern. This is because Visitor offers the ability to perform distinct and unrelated operations on a collection of types. Each operation becomes a concrete class derived from an abstract base; inside the operation, each type-specific processing unit becomes a virtual member function. The flow of operations ensures that the correct member function will be called depending on the actual type visited.

Consider applying Visitor to `Variant`. For a `Variant` instantiation, the Visitor pattern prescribes defining an abstract class that has a `visit` member function for each type that the `Variant` might contain. For example, for the `DatabaseField` type in Example 2 above, the following interface must be defined:

```
struct DatabaseFieldVisitor
{
    virtual void visit(int&) = 0;
    virtual void visit(double&) = 0;
    virtual void visit(std::string&) = 0;
    virtual void visit(Date&) = 0;
};
```

Each type that the `DatabaseField` might contain corresponds to a pure virtual function in `DatabaseFieldVisitor`. This means that for every `Variant` instantiation, the user must define a specific Visitor base class. The process, however, can be automated as shown in [3]. When using Loki's Visitor generic implementation, the `DatabaseFieldVisitor` above becomes:

```
typedef Loki::CyclicVisitor<void,
    TYPELIST_4(int, double, std::string, Date)>
    DatabaseFieldVisitor;
```

Loki's ability to define generic Visitor interfaces in terms of typelists provides the opportunity to embed the `typedef` above right inside `Variant`, so client code can readily use it:

```
template <...>
class Variant
{
public:
    typedef Loki::CyclicVisitor<void, TList>
        StrictVisitor;
};
```

To use the Visitor pattern as implemented by Loki, `Variant` must define an `Accept` function that takes a `StrictVisitor` as its only parameter, and dispatches it to the stored type. This is realizable by adding a new pointer to function to `vTable`, and forwarding to it from `Variant`'s `Accept` member function:

```
template <...>
class Variant
{
public:
    typedef Loki::CyclicVisitor<void, TList>
        StrictVisitor;
    void Accept(StrictVisitor& visitor)
    {
        (vptr_->accept_)(*this, visitor);
    }
};
```

The function implementation that's bound to `accept_` simply calls the appropriate `visit` overload for the `StrictVisitor` object.

The resulting context is favorable to writing correct, deterministic programs that treat all the types that a `Variant` might contain. Users who derive from `Variant<...>::StrictVisitor` must implement all member functions; otherwise, a compile-time error will occur.

However, some users might desire a less restrictive visitation process without giving up the advantages of visiting.

For example, consider a database function that performs number crunching on a specific column in a query result. The only types that the function is interested in are of type `double` and `DatabaseNull`, a placeholder for NULL fields in relational databases. It would be awkward to implement do-nothing stubs just to satisfy the type system; in this case, a more flexible approach is needed.

The Acyclic Visitor, described in [6], is a solution to such a need. The Acyclic Visitor is a more flexible approach to visitation that allows a `Visitor` object to visit any subset of types. Acyclic Visitor is also described in [3], where a generic implementation is also provided. Using that implementation, we can enable acyclic visitation for `Variant` with ease:

```
template <...>
class Variant
{
    ...
public:
    typedef Loki::BaseVisitor
        NonStrictVisitor;
    bool Accept(NonStrictVisitor& visitor)
    {
        return (vptr_->acceptNonStrict_)(*this, visitor);
    }
};
```

The two methods of visitation coexist and don't interfere in any way. The non-strict version of `Accept` returns a Boolean value that is true if the type was actually visited.

Two more `Accept` overloads are added to complete the visitability feature of `Variant`: the `const` counterparts of the acceptors described above. Without them, it is impossible to visit constant `Variant` objects.

8. Variant-to-Variant Conversions

Once visitation is in place, a number of complex tasks become considerably simpler. Consider the following example where a conversion between `Variants` is needed:

```
typedef Variant<
    TYPELIST_4(int, double, std::string, Date)>
    DatabaseField;
typedef Variant<
    TYPELIST_3(unsigned int, unsigned short, std::string)>
    FilteredData;
    ...
DatabaseField dbField;
    ...
FilteredData myData(dbField);
```

The intent is to initialize one instantiation of `Variant` with another instantiation of `Variant`. This is possible if the source `Variant` (here, `DatabaseField`) contains a type that is convertible to a type in the destination `Variant` (here, `FilteredData`).

If, for example, `dbField` contains an `std::string`, `myData` should be initialized with that string. If `dbField` contains a `Date`, an exception will be thrown (assuming `Date` cannot be converted to a string or to an integral type). Finally, if `dbField` contains an `int`, an ambiguity exception will be thrown because the `int` can be converted to both `unsigned int` and `unsigned short`.

Consequently, the constructor that converts a `Variant` instantiation to another performs the following algorithm:

- If the source `Variant` type is present in the typelist of the destination `Variant` type, then that will be the type of the destination's content.
- If the source `Variant` type can be implicitly converted to exactly one type in the typelist of the destination `Variant` type, then that conversion will be performed.
- In any other case, an exception is thrown.

This potentially slow algorithm can be solved elegantly and efficiently through visitation. The target `Variant`'s constructor visits the source `Variant` instantiation with a "converting visitor". In its `Visit` member function, `ConvertingVisitor` performs a compile-time lookup and dispatch for initializing the

target `Variant` in constant time. In the case of an ambiguity or a nonexistent conversion, an exception is thrown.

The `ConvertingVisitor` is generated using `Loki::GenLinearHierarchy` [3], a library facility that enables full hierarchy generation. In this case, the hierarchy is needed for implementing all the `Visit` overloads. The mechanism used for deciding type compatibility is, again, `Loki::Conversion`.

9. Variants Supporting Unbounded Types

Let's revisit `Variant`'s templated constructor:

```
template <...>
class Variant
{
public:
    template <class T>
    Variant(const T& val)
    {
        STATIC_CHECK((Loki::TL::IndexOf<TList, T>::value >= 0),
            Invalid_Type_Used_As_Initializer);
        new(&buffer_[0]) T(val);
        vptr_ = &VTableImpl<T>::vTbl_;
    }
};
```

The belonging of `T` to the set of types accepted by `Variant` is enforced only by the `STATIC_CHECK` invocation in the constructor. Nothing prevents a `Variant` from accepting any type whose size and alignment are compatible with `buffer_` and `Align`.

Such a policy would open the door to many interesting applications. The type is not limited to a well-defined set anymore; you can store any type that satisfies the size and alignment constraints in a `Variant`, including types that were not considered at the time of defining the `Variant` specialization of choice. If `Variant`'s size is big enough to accommodate pointers or smart pointers, `Variant` becomes an universal data packager.

To accommodate unbounded types, we only have to add an extra parameter type to `Variant`: a flag that tells whether the `Variant` must be bounded or not. The class definition and the constructors become:

```
template <class TList,
    bool unBounded = false,
    class Align = AlignmentCalculator<TList>::Result>
class Variant
{
public:
    template <class T>
    Variant(const T& val)
    {
        STATIC_CHECK(
            unBounded && sizeof(T) <= sizeof(buffer_) ||
            (Loki::TL::IndexOf<TList, T>::value >= 0),
            Invalid_Type_Used_As_Initializer);
        new(&buffer_[0]) T(val);
        vptr_ = &VTableImpl<T>::vTbl_;
    }
};
```

This approach combines the advantages of bounded and unbounded discriminated unions: you can still benefit of the amenities provided for the types in the `typlist`, but you also can store types that are completely foreign to the passed-in `typlist`.

If you're not interested at all in static knowledge about the type contained, but would like to store objects up to a certain size, you can use `char[N]` as the only type of `TList`. For example, the following `typedef` defines a `Variant` that can store any type that's up to 64 bytes in size:

```
typedef Variant<TYPELIST_1(char[64]), true> Any;
```

Non-strict visitation described in Section 7 above becomes essential for ensuring proper, elegant decoherence of unbounded `variant` instantiations.

10. Conclusion

This paper introduced a completely generic implementation of discriminated unions. The array of features exposed by this implementation makes it very flexible and suitable to a large range of applications.

`variant`'s essential characteristics are the ability to specify the set of possible types, seamless support for primitive and user-defined types, high efficiency through use of function pointers and avoidance of free store allocation, a rich decoherence model, and a powerful conversion system.

11. Acknowledgements

I would like to thank Thant Tessman for a thorough review full of inspiring insights.

Bibliography

- [1] F. Cacciola. "An Improved Variant Type Based on Member Templates" (C/C++ Users Journal, October 2000)
- [2] Volker Simonis "Chameleonic Objects" (C++ Report, January 2000)
- [3] A. Alexandrescu. "Modern C++ Design" (Addison-Wesley, 2001)
- [4] H. Sutter. "Exceptional C++" (Addison-Wesley, 2000)
- [5] E. Gamma, et al. "Design Patterns" (Addison-Wesley, 1994)
- [6] R. Martin, et al. "Pattern Languages of Program Design" (Addison-Wesley, 1997) p. 93
- [7] K. Czarnecki, U. Eisenecker. "Metalisp" (<http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm>, 1998)
- [8] K. Czarnecki. "Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models" (Ph.D. Thesis, University of Ilmenau, Germany, 1998, <http://www.prakinf.tu-ilmenau.de/~czarn/diss>)
- [9] K. Czarnecki, U. Eisenecker. "Generative Programming" (Addison-Wesley, 2000)