Some Basic Data Structures and Algorithms for Chemical Generic Programming

Wei Zhang, Tingjun Hou, Xuebin Qiao, and Xiaojie Xu*

College of Chemistry and Molecular Engineering, Peking University, Beijing 100871, China

Received February 15, 2004

Here, we report a template library used for molecular operation, the Molecular Handling Template Library (MHTL). The library includes some generic data structures and generic algorithms, and the two parts are associated with each other by two concepts: *Properties* and *Molecule*. The concept *Properties* describes the interface to access objects' properties, and the concept *Molecule* describes the minimum requirement for a molecular class. Data structures include seven models of *Properties*, each using a different method to access properties, and two models of molecular classes. Algorithms include molecular file manipulation subroutines, SMARTS language interpreter and matcher functions, and molecular OpenGL rendering functions.

INTRODUCTION

Generic programming (GP)[1] is a new programming paradigm supported by the C++ programming language. Programs using generic programming include generic data structures (in the form of template classes) and generic algorithms (in the form of template functions). Data structures and algorithms are associated with each other by "concept". A concept describes a set of requirements on a data type, and when a specific data type satisfies all of these requirements, we say that it is a "model" of that concept. Actually, algorithms operate on models of its argument concept, and data structures are written to be models of algorithms' argument concepts. This "concept" abstraction is the essence of generic programming. GP is different from the object-oriented (OO) paradigm,[2] which involves associating data types with a hierarchy of inheritance in many aspects.

Generic programming has achieved great progress in past years. The STL (C++ Standard Template Library),[3] as the first commercial product of generic programming, was accepted in 1994 as a part of the C++ standard.[4] Many basic components of C++ programming language, like string and stream, have been rewritten using generic programming.

As for chemical software development, GP has been used for the design of scientific computing programs,[5−7] but there is no library for generic programming that provides some common functions. A template library, Molecular Handling Template Library (MHTL, Figure 1), has been designed for this requirement. It includes some generic data structures and generic algorithms, and these two parts are coupled together by two concepts: *Properties* and *Molecule*. The concept *Properties* describes the interface to access objects' properties, and the concept *Molecule* describes the minimum requirement for a molecular class. Data structures include seven models of Properties, each using a different method to access properties, and two models of molecular classes. Algorithms include molecular file manipulation subroutines, SMARTS language interpreter and matcher functions, and molecular OpenGL rendering functions. The complete version of MHTL consists of approximately 6000 lines of code in C++ programming language. All calculations experiments were carried out on a PC. The program has been tested on IRIX, Linux, and Windows operating systems, and the source codes can be obtained freely from the authors upon request. The two parts of MHTL are discussed in the following two sections, respectively.

* Corresponding author phone: (86)-10-62757456; fax: (86)-10-62751708; e-mail: xiaojxu@chem.pku.edu.cn.

DATA STRUCTURE

The data structure part of MHTL includes some classes, which are mainly models of two concepts: *Properties* and *Molecule*.

Concept *Properties* and Its Models. A common action performed on a chemical object is to set and get its property. The property can be the atom's position and force, bond's order, molecule's boiling point, and melting point. As it can be seen, properties can be of any data type, even user-defined types. This uncertainty in property type makes its access and storage a difficult job, and in GP the solution to this type of uncertainty is to set the type as template argument; thus we require that models of *Properties* have the following two member functions:

```
template< typename Ptype > void SetProperty( const Ptype& value );
template< typename Ptype > Ptype GetProperty();
```

These two functions can be used to access the property of any type, but properties of the same type cannot be distinguished, and it requires that users declare each property a data type explicitly.

To help users define property types easily, we have defined a template class PropertyT as follows:

```
template< int Pid, typename Ptype >
class PropertyT
{
public:
        PropertyT() {};
        PropertyT(const Ptype & rhs ):m_value(rhs){ };
        operator Ptype() const { return m_value; }
        static const int PropertyId = Pid;
private:
        Ptype m_value;
};
```

10.1021/ci049938s CCC: $27.50
Published on Web 06/17/2004

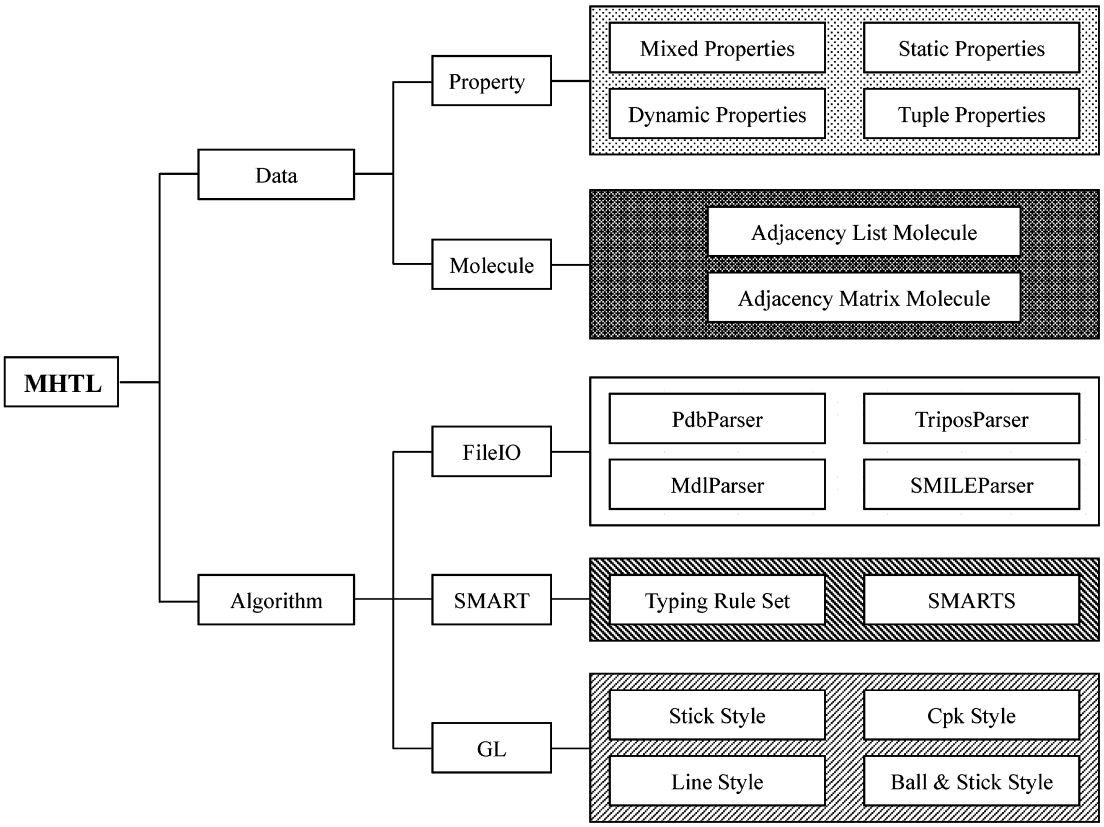

Figure 1. The schematic representation of MHTL.

PropertyT takes two template arguments: one is Pid, which is the identification number of the property type, and can be used to distinguish different properties; the other is Ptype, the actual data type of the property. To ensure the behavior of the derived type is the same as the actual type, a copy constructor and a data type conversion function have been included in the class. A static and const data member named PropertyId has been defined inside the class, which is used for the identification of property type. We defined a trait for property to help users access PropertyId. The trait follows:

```
template< typename PropertyType >
struct PropertyTraits
{
    static const int PropertyId = PropertyType::PropertyId;
}
```

Sometimes users need to declare their property type without using PropertyT, and they may not be able to declare PropertyId inside the data type; however, they can declare PropertyId by specializing PropertyTraits such as this:

```
template< >
struct PropertyTraits< UDT >
{
    static const int PropertyId = 10;
}
```

Type UDT is assigned a PropertyId of 10 in this way, and users should take care of the uniqueness of PropertyId by themselves.

Since the concept *Properties* has been defined and the property types have been declared, our problem is how to realize these two member functions.

Traditionally, property is stored as a member variable and accessed using the setter and getter function. This method is very efficient, but in this way properties are not accessed by a unified interface. The following code illustrates how this method can get access to the atom's element:

```
class TraditionalProperties
{
public:
    void SetElement( int element )    { m_Element = element; }
    int GetElement( void )            {   return m_Element; }
private:
    int m_Element;
};
```

The technique of template specialization can be used to improve the traditional method. Using this method, we can realize a class that is the model of *Properties* and stores property as a member variable. The following code demonstrates this technique:

```
class TsProperties
{
public:
    template< typename Ptype > void SetProperty( const Ptype& value )    { }
    template< typename Ptype > Ptype GetProperty( void ) { return Ptype(); }
private:
    Element m_element;
};
template<> void TsProperties::SetProperty(const Element& value ) { m_element = value; }
template<> Element TsProperties::GetProperty( void ){ return m_element; }
```

When we are using the template specialization method to access property, we still need to define two member functions for each property. This work can be simplified by using template meta-programming.[8]

Template meta-programming is a compile-time language, which means programs written in template meta-programming run and get their results in compile-time. The MPL module of the BOOST library is a product of template meta-programming.[9]

The core component of MPL is type list. People can first construct a type list and then perform many kinds of operations on it, such as append, remove, find, and so on. The tuple[10] component of MPL is appropriate for use here, which is generated by linear inheritance of tuple field. The usage of tuple is like struct in C language, but it provides a unified interface to access its member by type. The following code illustrates the tuple method:

```
template< typename Typelist >
class TuplePropertiesT
{
public:
    template<typename Ptype > void SetProperty( const Ptype& value )
    {
        field< Ptype > ( m_Properties ) = value;
    }
    template<typename Ptype>        Ptype GetProperty()
    {
        return field< Ptype >( m_Properties );
    }
private:
    typename mpl::inherit_linearly< Typelist,
                                    mpl::inherit< _1, tuple_field<_2> >
                                  >::type m_Properties;
};

typedef TupleProperties< mpl::list< element > > ThisProperties;
```

The last line declares a class that is equal to TsProperties in usage.

The three property access techniques we discussed before (traditional method, template specialization method, and tuple method) are all under such an assumption that we known the property will be accessed. If the type of property is unknown, these three methods cannot be used, so we need some property access mechanism that can convert property of any type into a media and can convert it back whenever we need.

Several methods can be used to establish such a mechanism: first, we can use malloc() to allocate memory and use free() to release memory. The property is stored as void pointer, and the void pointer can be converted back to property by static cast. The following code illustrates this method:

```
class StaticProperties{
public:
    StaticProperties() {}
    virtual ~StaticProperties()
    {
        std::map< int, void*>::iterator i = m_Pmap.begin();
        for( ; i != m_Pmap.end(); i++ )
            free( i->second );
    }

    template <typename Ptype> void SetProperty(const Ptype& value)
    {
        void* ptr = malloc( sizeof(Ptype) );
        new(ptr) Ptype(value);
        m_Pmap[ Ptype::PropertyId ] = ptr;
        m_Pmap[ Ptype::PropertyId ] = new Ptype(value);
    }

    template <typename Ptype> Ptype GetProperty( )
    {
        return *(Ptype*)( m_Pmap[ Ptype::PropertyId ] );
    }
private:
    std::map< int, void* > m_Pmap;
};
```

Since the static method is not type safe, we need some alternation. We can convert property into string by using the "≪" operator and can convert it back by using the "≫" operator. In the following code, we define a class StringProperties and use this idea to access properties. The class StringProperties has a very interesting feature, which supports many kinds of conversion.

Table 1. The Comparison between Property Access Methods

		property access method				
compiler setting	property name	traditional	template specialization	tuple	string	dynamic
Level-3 optimized	element	2	2	2	11 925	262
	partial charge	3	4	4	16 762	272
	atom name	745	1737	1310	14 617	1765
	position	73	272	210	36 816	514

```
class StringProperties
{
public:
    template <typename Ptype>
    void SetProperty(const Ptype& value)
    {
        int pid = PropertyTraits<Ptype>::PropertyId;
        std::ostringstream oss;
        oss << value;
        m_Pmap[ pid ] = oss.str();
    }
    template <typename Ptype>
    Ptype GetProperty( )
    {
        int pid = PropertyTraits<Ptype>::PropertyId;
        if( m_Pmap.count( pid ) )
        {
            std::istringstream iss( m_Pmap[ pid ].c_str() );
            Ptype value;
            iss >> value;
            return value;
        }
    }
private:
    std::map< int, std::string > m_Pmap;
};
```

This string method is obviously slow, so we keep looking for a faster method. We can use pointer's dynamic cast, which is a new feature of the C++ programming language. To realize property in this method is difficult, but we do not need to do it. The "any" module of the boost library has realized it for us.[9] The class DynamicProperties defined in the following code illustrates how to construct a model of *Properties* using dynamic cast of pointer.

```
class DynamicProperties
{
public:
    template <typename Ptype> void SetProperty(const Ptype& value)
    {
        int pid = PropertyTraits<Ptype>::PropertyId;
        m_Pmap[ pid ] = value;
    }
    template <typename Ptype> Ptype GetProperty( ) const
    {
        int pid = PropertyTraits<Ptype>::PropertyId;
        if( m_Pmap.count( pid ) )
        {
            return boost::any_cast<Ptype>( m_Pmap.find( pid )->second );
        }
    }
private:
    std::map< int, boost::any > m_Pmap;
};
```

Until now, we have introduced six methods to access property. These methods can be classified into two categories: direct method and indirect method. In these methods, the static cast method is worst because it is not type safe, and we do not recommend that users use this method. The performances of the remaining five methods have been compared. The comparison was done on four kinds of properties: element (integer number), partial charge (floating number), atom name (std string class), and position (Vector3d). The compiler used here was GNU C++ compiler 3.2, and the code was compiled under the optimization on Level 3. The results are illustrated in Table 1. From Table 1, we have found that: (1) the traditional way is the fastest in every case; (2) for properties of basic types such as *double* and *int*, the tuple method and the template specification method can be as efficient as the traditional method, but for properties of complicated classes, these two methods are much slower than the traditional method; and (3) the direct access methods are much faster than the indirect methods.

Although the direct access method is much faster than the indirect method, in some programs we still need to use the indirect access methods. We note here that these two kinds of methods could be combined together to form a data type that uses the direct access method to some predefined properties and uses the indirect access method to access other properties. In the following code, we define such a template MixedPropertiesT, which takes three arguments. The first

```
template< typename typelist, typename DirectProperties, typename IndirectProperties >
class MixedPropertiesT
{
public:
    template< typename Ptype >
    struct ChooseProperties
    {
        typedef typename mpl::find< typelist, Ptype >::type iter;
        typedef typename mpl::end< typelist >::type end_iter;
        typedef typename mpl::if_< boost::is_same< iter, end_iter >,
                                   IndirectProperties,
                                   DirectProperties >::type    type;
    };
    template<typename Ptype >
    void SetProperty( const Ptype& value )
    {
        typedef ChooseProperties< Ptype >::type Properties;
        field< Properties >( m_Properties ).SetProperty( value );
    }
    template<typename Ptype>
    Ptype GetProperty()
    {
        typedef ChooseProperties< Property >::type Properties;
        return field< Properties >( m_Properties ).Get<Ptype>();
    }
private:
    typename mpl::inherit_linearly< mpl::list< SpecializedProperties, OtherProperties >,
                                    mpl::inherit< _1, tuple_field<_2> >
                                  >::type m_Properties;
};
```

is typelist, which is a list of property types that should be accessed using the direct access method; the second is DirectProperties, the type name of direct access properties;

the last one is IndirectProperties, the type name of the direct access properties. In the realization of MixedPropertiesT, we used the meta-function **find** to find out if a specified type is in a type list, which is imported from the boost's mpl module.

Concept *Molecule* and Its Models. Molecule is the common object that chemical algorithms may operate on. Generally, a molecule contains several atoms, and atoms are associated with each other by bonds; meanwhile, molecule, atoms, and bonds have their own properties. Moreover, molecules can be classified into monomers and polymers, and a polymer is normally made up of several residues.

In MHTL, the *Molecule* concept requires data types to realize the following member functions:

```
int NumberAtom();
template<typename Ptype> void SetAtomProperty(int index, const Ptype& value );
template<typename Ptype> Ptype GetAtomProperty( int index );
vector<int> GetBondsOfAtom( int index );
int GetBondBetweenAtom( int index_a, int index_b );

int NumberBond();
template<typename Ptype> void SetBondProperty(int index, const Ptype& value );
template<typename Ptype> Ptype GetBondProperty( int index );
int GetBondBeginAtom( int bond_index );
int GetBondEndAtom( int bond_index );

template<typename Ptype> void SetProperty(const Ptype& value );
template<typename Ptype> Ptype GetProperty( );
```

As for the realization of molecule data type, many methods can be used. In these methods, adjacent list and adjacent matrix are mostly used. The adjacent list method involves saving bond associations in a bond list, while the adjacent matrix method involves saving them in a 2-D matrix. The two methods both have their advantages and disadvantages. The adjacent list method is quick in counting the bond number, and getting the bond's atom, while the adjacent matrix method is quick in getting the bond of atoms.

ALGORITHM

Algorithms in MHTL include the following:

(1) Molecule input and output subroutines using different file formats, which are all template functions and take *Molecule* as their argument concept. We are now providing the manipulation for files in MDL MOL,[11] PDB,[12] Daylight SMILES,[13] and Sybyl Mol2[14] format.

(2) SMARTS[13] language interpreter and matcher functions. SMARTS language is a pattern language designed for the description and matching of chemical environments and has been employed by many chemical softwares. SMARTS libraries that can be obtained are mostly C++ class libraries, so we wrote a SMARTS module using GP.

(3) Molecule 3d-rendering subroutines using OpenGL library. Molecular 3d-rendering using OpenGL[15] is necessary for GUI of most chemical softwares, and we provide a module in MHTL that helps chemists realize this function easier.

CONCLUSION

Generic programming is a kind of useful programming paradigm supported by many programming languages, while its usage in computational chemistry is limited. We have tried to import it into chemical software development. We defined two commonly used concepts and defined many models of these concepts on the basis of different policies, and we wrote many algorithms operating on the concept. These algorithms have three basic functionalities: molecular file input and output in different file formats, atom pattern recognition using SMARTS language, and molecular 3D-rendering using OpenGL. We hope these algorithms can help chemists build their software easier and more quickly.

ACKNOWLEDGMENT

This project is supported by the National Natural Science Foundation of China (NSFC 29992590-2 and 29873003).

REFERENCES AND NOTES

(1) Austern, M. H. *Generic Programming and the STL*, 1st ed.; Addison-Wesley Pub. Co.: New York, Oct. 1998.
(2) Booch, G. *Object-Oriented Analysis and Design with Applications*, 2nd ed.; Addison-Wesley Pub. Co.: New York, Oct. 1993.
(3) Stepanov, A.; Lee, M. *The Standard Template Library*; Hewlett-Packard Laboratories, 1994.
(4) ISO/IEC 14882. *Programming Languages-C++*, 1st ed.; ISO/IEC, Sept. 1998.
(5) Gerlach, J.; Kneis, J. Generic Programming For Scientific Computing in C++, Java (TM), and C#, Advanced Parallel Processing Technologies, Proceedings. *Lect. Notes Comput. Sci.* **2003**, *2834*, 301−310.
(6) Kettner, L. Using Generic Programming For Designing A Data Structure For Polyhedral Surfaces. *Computational Geometry-Theory And Application*; Elsevier: New York, 1999; Vol. 13, pp 65−90.
(7) Trobec, R.; Sterk, M.; Praprotnik, M.; Janezic, D. Parallel Programming Library For Molecular Dynamics Simulations. *Int. J. Quantum Chem.* **2004**, *96*, 530−536.
(8) Alexandrescu, A. *Modern C++ Design*, 1st ed.; Addison-Wesley Pub. Co.: New York, Feb. 2001.
(9) Gurtovoy, A.; Abrahams, D. *The BOOST C++ Meta Programming Library*; http://www.mywikinet.com/mpl/paper/mpl_paper.pdf.
(10) Jarvi, J. Tuple Types and Multiple Return Values. *C/C++ Users Journal*; 2001; Vol. 12, 2nd paper.
(11) *CTFile Format*; MDL Co., Aug. 2002; http://www.mdli.com.
(12) *PDB Format Description Version 2.2*; The Research Collaboratory for Structural Bioinformatics(RCSB), Dec. 1996; http://www.rcsb.org/pdb/docs/format/pdbguide2.2/guide2.2 frame.html.
(13) James, C. A.; Weininger, D.; Delany, J. *Daylight Theory Manual*; Daylight Chemical Information Systems Inc., Jun. 2003; http://www.daylight.com/release/manuals.html.
(14) *Tripos Mol2 File Format*; Tripos Inc.; http://www.tripos.com/custResources/mol2Files/index.html.
(15) Segal, M.; Akeley, K. *The OpenGL Graphic System: A Specification (Version 1.5)*; Silicon Graphic, Inc., 2003; http://www.opengl.org/documentation/specs/version1.5/glspec15.pdf.

CI049938S