Safe and efficient data types in C++

Nicolas Burrus

Technical Report n^o0212 - January 2003

Using C++ builtin types is very unsafe. Indeed, they are inherited from C types, which do not have overflow checking and have dangerous side effects and unexpected behaviors. Using intensive meta programming, it becomes possible to design safe data types with a minimal runtime overhead. As we want to be able to use existing algorithms, these types have to interact transparently with C++ builtin types. Primarily designed for Olena, a generic image processing library, our work provides mechanisms to allow easy integration in generic algorithms.

Keywords
Data types, Object-Oriented Programming, Scientific computing, Generic Programming, Static Meta-programming, Advanced C++

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
lrde@epita.fr – http://www.lrde.epita.fr

Contents

Chapter 1

Introduction

1.1 Data types?

By data types, we mean data storage unity and operations for their manipulation. This includes Integers, Floats, Complexes, but also Vectors and Matrices, etc. Most of these data types are usually implemented in the programming language's core, called builtin types.

For example, we can distinguish four categories of data types :

- Scalars (Integers, Floats).
- Complex numbers (with polar representation, rectangular representation).
- Enumerated types (boolean, labels).
- Vectorial types (vectors, matrices).

1.2 Programming language

In this report, we use the C++ programming language to implement data types. Paradigms and programming principles described here may be applied partially or totally with another language, but it has to implement enough static features to keep the result fast and interesting.

All the provided source code tries to stay compliant with C++ standard, and always refuses compilers extensions. The code has been tested with g++-2.95, g++-3.0, g++-3.1, g++-3.2, comeau v4.301, and intel C++ compiler v7.

1.3 Context

Our data types were initially developed for Olena [**?**, **?**, olena] an image processing generic library in C++ developed by the LRDE (EPITA Research and Development Laboratory). Olena uses intensive C++ meta programming and tries to give maximal genericity.

An image is represented by a matrix of points, and a point is represented by a data type. One of the goals of the library is to provide a really efficient way to compute image processing algorithms, so our data types need to be as efficient as possible. Furthermore, the library wants to provide safe algorithms, and wants to detect types overflow, arithmetic errors, and incompatible casts when possible, to avoid common mistakes and make safer programs.

1.4 Motivations

1.4.1 Builtin types are dangerous

Problems start with the following statement : C++ builtin types are not safe. Consider the following code :

```
int i = 256;
unsigned char foo = i; // foo == 0
```

The C++ compiler will accept this code without any warning, and at runtime the program will execute without any warning, but foo will be 0 and not 256. It is certainly a programmer mistake, but she won't be noticed of the problem.

Arithmetic operations too have potential problems :

```
unsigned int i = UINT_MAX;
unsigned int j = 5;
unsigned long long k = i + j; // k == 4
```

Here k is big enough to contain to result of `i + j`, but in C++, an addition of two **unsigned int** is stored on an **unsigned int**, before being casted into **unsigned long long**. That's why an overflow occurs.

We really want safer operations, and much stronger type checking!

1.4.2 Generic algorithms

Olena is a generic library, we need to write generic algorithms. This implies some new constraints, here is an example:

```
template <class DataType>
ResultType Sum(const Image<DataType>& ima)
{
  ResultType s = zero_for_ResultType();
  Image::iter i(ima);

  for_all(i)
  s += ima[i];
  return s;
}
```

- DataType is generally too small to store the sum, so we need to specify a bigger type for ReturnType, depending on Datatype.
- zero_for_result is also dependent on ReturnType, the function must not return the same type if we are manipulating integers or vectors.

That's why C++ builtin types do not satisfy our needs. Nevertheless they have a big advantage: they are fast, and their operations can be easily optimized by the compiler.

Many existing projects are confronted to similar problems with types, so let us make a tour of interesting existing approaches.

Chapter 2

Overview of existing approachs

2.1 Octave, Matlab

2.1.1 Presentation

Octave [2] and Matlab [1] use only double precision numbers as core data representation. They doesn't support integers, shorts, chars, etc.

To allow constraints on data types, Octave introduces a "range" structure, defined by its first element, an increment (1 by default) and a maximal value.

2.1.2 Benefits

- Avoid problems with type range choices.
- Resolve genericity problems with basic data types.
- Solve overflow problems.
- Very simple to use.

2.1.3 Drawbacks

- Double precision numbers take 3 times more memory than an integer, and 12 times more than a char. So when loading a 3D grey level image in memory, it takes 1728 times more memory with octave/matlab than with the equivalent in C++ using char.
- Noticeable overhead in CPU, as floats are quite slower than integers.

2.2 Vigra

2.2.1 Presentation

Vigra [4] underlies on C++ builtin types, and defines traits to allow generic algorithms. Traits can be interpreted like a structure which associates types or functions to a specific type. Here is an example :

```
template <class T> struct larger_traits;

template <> struct larger_traits<char> {
  typedef short larger_type;
};
```

```
  template <> struct larger_traits<float> {
    typedef double larger_type;
  };
```

These traits define the associated larger type for **char** and **float**. Then larger_traits <T>::larger_type will return the good larger_type associated with T.

So let us see how our generic sum is implemented with Vigra:

```
template <class DataType>
typename vigra::NumericTraits<typename DataType>::Promote
Sum(const Image<DataType>& ima)
{
  typedef typename vigra::NumericTraits<typename DataType>::Promote ResultType;
  ResultType s;
  Image::iter i(ima);

  s = vigra::NumericTraits<ResultType>::zero();
  for_all(i)
   s += ima[i];
  return s;
}
```

Here return type is determined by vigra::NumericTraits. Traits can also help writing generic arithmetic operations and other things which need to access properties associated to one or several types.

2.2.2 Benefits

- Vigra uses builtin C++ types, so it doesn't have compatibility problems with existing algorithms.
- The system is simple, and does not improve too much compilation time.
- No overhead at execution time.

2.2.3 Drawbacks

- As it uses builtin types, it cannot implement safe arithmetical operators.
- Designing secure code implies a heavy work by the user.
- Non extensible, the set of data types is predefined.

2.3 Olena before v0.6

2.3.1 Presentation

Note : in this section, *Olena* refers to Olena versions older than 0.6.

Olena defines its own data types, int_u<nbits> (unsigned integer on nbits), int_s<nbits> (signed integer on nbits), and so on. Thus it has a total control on side effects with these types. This allows implementation of various verifications, increasing program safety.

Moreover, Olena defines traits in each of its type, so that writing generic algorithm is possible.

For each type, a coercion operator is defined toward the closer builtin C++ type to allow compatibility with external algorithms and programs using builtin types.

To ensure safety, Olena mostly uses static checks. This means everything needs to be as strongly typed as possible to determine at compile time most of the insecure operations. This is why Olena implements type growing, for example, an addition of two int_u<8> results in an int_u<9>, thus it is possible to statically check that assignements will not imply overflows.

2.3.2 Benefits

- No overhead at execution (everything static).
- A program which compiles does not have problems at execution.
- The types can be used in external algorithms as they can convert into corresponding C++ builtin types.

2.3.3 Drawbacks

- Some cases escape from static checking, for example with int_u8 a += b; it cannot check statically if a will overflow or not.
- Static checking often implies annoying behaviors, as shown in the following code :

```
// int_u8::min() + 1 returns int_u9, so we need to cast it
const int_u8 init = cast :: force<int_u8>(int_u8::min() + 1);

template <class T>
T average(const list<T>& l)
{
  T::larger_type sum = T::larger_type::zero();

  for (list <T>::iterator i = l.begin (); i != l.end(); ++i)
    sum += i;

  // Here we know sum / l.size() fits in T as it's an average, but we
  // have to insert a cast::force.
  return cast::force<T>(sum / l.size());
}
```

- Generic algorithm cannot be applied to builtins, as they don't have associated traits. (T::larger_type does not work if T is **int**).
- Template tricks implies heavy compilation time overhead.

2.4 Ada

2.4.1 Presentation

The Ada programming language introduces interesting features for type checking and programs security. Ada integer type has a strict interval, this means assigning too large a value into an integer will fail at execution (or at compilation time if there are already enough informations). But Ada also defines the notion of "subtypes" and constraint, which allow declaration of a type which can take its value into 0..10 for example, and dynamic checks will be done if needed. It even allows dynamic ranges for subtypes, calculated from expressions.

2.4.2 Benefits

- Programs will fail at execution if there is any problem, avoiding implicit overflows and other common problems.
- The notion of range is really nice for programmers.

2.4.3 Drawbacks

- Type restrictions are too strong so explicit casts are often required.
- Runtime overhead because of dynamic checks.

2.5 Conclusion

This overview present several interesting features :

- Generic algorithms handling, including builtin types (Vigra).
- Dynamic checks and range constraints (Ada).
- Strong typing with growing types and static checks (Olena < 0.6).

However, none of the above implementation satisfy all our needs, so we will just take these features as a starting point, and now focus on what we really want and need with data types.

Chapter 3

New datatypes : goals and features

This chapter gives an overview of the objectives we will try to reach. It is essentially a list of features and constraints we will keep in mind in the next chapter.

3.1 Data type ?

There are many C++ builtin types: unsigned int, signed int, unsigned short, signed short, signed char, etc. We do not want so many different types representing the same fundamental one, integer. Family of types like unsigned integers, signed integers, floats would be much simpler to manipulate.

3.2 Safety

Data types should detect most of the programmer low-level mistakes. Indeed, as they have all control about operations concerning them : assignments, arithmetic calculus, coercion, comparisons, ..., they can ensure that results are coherent and without overflows. Using these possibilities, a program which have not explicitly detected any problem should always have right values.

To perform such controls, our implementation will have to insert check code before every assignments and operations. At a first glance, this seems quite incompatible with our second main goal, efficiency.

3.3 Efficiency

Our types will be used to compute scientific calculus, so they have to be fast ! Fortunately, strong typing and genericity will help. Indeed, many operations do not need any check at runtime if we already know the value domains of operands are compatible. Here are basic examples of assignments and operations which do not need runtime checks :

```
Integer_on_8_bits = Integer_on_less_than_8_bits;
Integer_on_9_bits = Integer_on_8_bits + Integer_on_8_bits;
```

So we want our data types to make an optimal use of static information which can be available, in order to keep maximal efficiency.

3.4 Strong typing

To gather maximal information at compile time about type ranges and properties, it is necessary to have a type system as strongly typed as possible. This permits the disabling of useless dynamic checks and give the user a way to keep his types as small as possible. This is especially interesting for image processing, to keep an entire image in the minimal memory space.

3.5 Generic algorithms compatibility

As explained in the introduction, being compatible with generic algorithms is not immediate. We want both our types and builtin types to be compatible with generic algorithms.

3.6 Controlled interactions

Of course we do want usual operations and conversions between types. This means unsigned integers should have arithmetic operations with other integers, unsigned integers should be able to convert into signed integers and vice versa if the signed integer is positive.

3.7 Builtin interactions

More than just allowing conversion between our self-defined types and their closer builtin types, it would be appreciable to allow arithmetic operations between them, *without* heavy code rewriting.

3.8 Decorations

Common types have well defined properties. For example, a vector type does not have comparison operators. But in some cases, for a particular use, the user may want to have a relational operator on vectors. The type with comparison operator should behave exactly the same way a vector does, except for comparison. Obviously we do not want to completely rewrite a new type, but just "decorate" the exiting vector type with a comparison operator.

3.9 Extensibility

More data types are always needed. Thus, implementing a new type should be kept as simple as possible. This also implies the type hierarchy to stay "open", allowing easy integration.

Chapter 4

Implementation

4.1 Introduction

This chapter describes the design and the implementation of the global data type system we realized. For a better understanding the next sections will only refer to scalar types, so let us describe them :

int_u<nbits, behavior> Unsigned integer on nbits. Behavior indicates what to do when a problem occurs, such as an overflow.

int_s<nbits, behavior> Same type than int_u but for signed integers.

sfloat Float numbers with simple precision.

dfloat Float numbers with double precision.

range<T, interval, behavior> range decorates a type T modifying its value domain. interval defines the new interval. behavior specifies the consequences of an overflow.

cycle<T, interval> cycle allows modulo calculus on an existing type T. Assignments will be applied modulo the interval given.

Three behaviors are given with the standard distribution :

unsafe Just ignore overflows. This is mainly for internal use.

strict An overflow aborts the program.

saturate If a value can't fit in the type interval, round it to the nearest bound.

To allow easy classification and discrimination on data types, they are organized in a hierarchical way. Actually, there are three hierarchies, as it is discussed on next section.

4.2 Global organization

4.2.1 Overview

The outline of our implementation mainly rely on orthogonalization. This approach gives much more modularity and integration possibilities. Types are organized by a hierarchical way. To get maximal performances, many static meta programming tricks are used, in particular static hierarchies [6]. Three hierarchies are defined :

Data hierarchy The main one. Used for value storage.

Characteristics hierarchy Associates types with data types. For example, cumul_type or larger_type.

Implementation hierarchy Defines operators and methods related to data types.

This division has several advantages :

- Builtins integration. Although they cannot integrates the data hierarchy, they can be represented in both characteristics and implementation hierarchies. Thus, operators implementation can be given for builtins (such as min() and max()), allowing generic algorithm to work with builtin types.
- More flexible source code, and help to implement static hierarchies.
- Decorators can directly inherit the implementation of a type, without necessarily inherit from the type. This results in a good factorization of the code.

These points will become more obvious when concrete implementation in C++ is discussed.

4.2.2 Data hierarchy

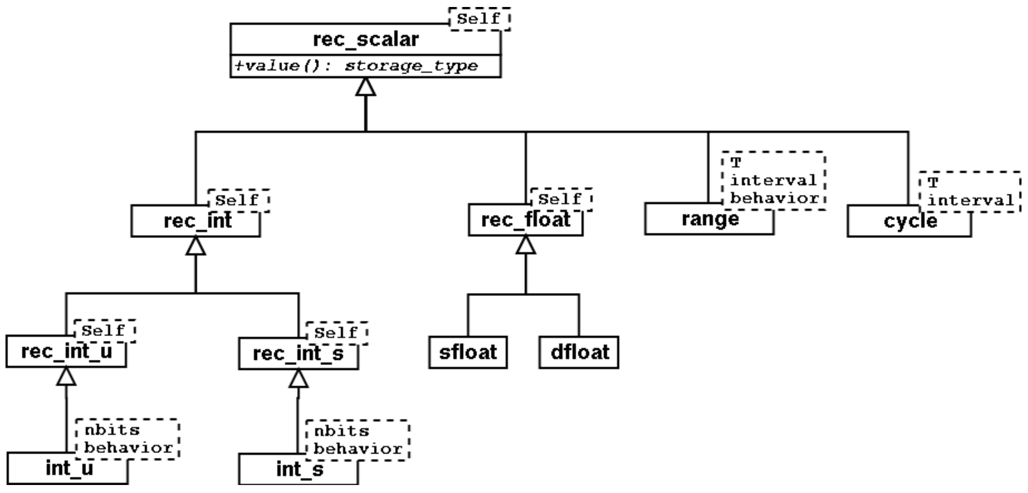

Figure 4.1: Data hierarchy.

Here is the scalar part of the data hierarchy. It is mostly interface classes, as no real methods are implemented into it, excepted constructors, conversion operators and the special value() method to retrieve the actual value of the type. The basic type used to store value is given by the characteristics associated to the corresponding type (for example, **unsigned char** is associated to int_u<8>.

4.2.3 Type characteristics hierarchy : typetraits

This hierarchy has the same structure than the data one. But this time, only typedefs are defined and inherited. As it is implemented using traits [6], builtins can be integrated. Here are example of such classes :

```
// Traits to determine the C builtin type big enough to store N bits

template <>
struct Get_CType_for_Unsigned<8>
```

```
{
  typedef unsigned char return_type;
};

template <>
struct Get_CType_for_Unsigned<16>
{
  typedef unsigned short return_type;
};

// ...

// Trait for int_u

template <unsigned nbits, class behavior>
struct typetraits<int_u<nbits, behavior> > : public typetraits_unsigned_integer
{
  typedef int_u<nbits, behavior> self;

  typedef Get_CType_For_Unsigned<nbits>::return_type value_type;
  typedef int_s<nbits+1, behavior> signed_type;
  typedef int_u <32, behavior> largest_type;
};

// ...

// Trait for int, it works with builtins !

template <>
struct typetraits<signed int> : public typetraits_signed_integer
{
  typedef int self;

  typedef self value_type;
  typedef self signed_type;
  typedef unsigned int unsigned_type;
  typedef long largest_type;
};
```

4.2.4 Implementation hierarchy : optraits

optraits follows a hierarchy similar to the two others. Now we have value and traits for types, optraits uses traits to implement operators and others methods on data types. Let us look at an extract of the source code :

```
template <class Self>
struct optraits_int_u : public optraits_int<Self>
{
  // Get the storage type associated with Self
  // Notice the use of typetraits, so that Self can be a builtin type.
  typedef typename typetraits<Self>::storage_type storage_type;

  // min and max return storage_type (generally builtins for non
  // decorated types) for internal reason (the idea is : instantiation
  // of int_u for example requires calls to min() and max() to check
  // bounds, so if min and max instantiate themselves an int_u, we enter
  // in an infinite loop.
  static storage_type min ()
  {
    return 0;
  };

  // ...
};

template <class nbits, class behavior>
struct optraits<int_u<nbits, behavior> >
  : public optraits_int_u<int_u<nbits, behavior> >
{
  typedef int_u<nbits, behavior> self; // shortcut
  typedef typename typetraits<self>::storage_type storage_type;

  static storage_type max ()
  {
    // Get_Max_From_Unsigned<nbits> calculate max from bit number.
    return Get_Max_From_Unsigned<nbits>::ret;
  }

  // ...
};

// Builtin type can inherit implementation from its non builtin equivalent.
template <>
struct optraits<unsigned int> : public optraits<int_u<32, strict> >
{
  // implementation is inherited !

  // You may want to add or override some functionalities thought.
};
```

4.3 Operators

The process chain used for arithmetic operators gives a good overview of the possibilities of our type organization. Many arithmetic operations are available between every scalar types, and the code often remains the same. This section introduces the problems with code factorization, and brings a working solution.

4.3.1 Problems

We would like to write something close to this :

```
template <class T1, class T2>
return_type operator+ (const rec_scalar<T1>& lhs, const rec_scalar<T2>& rhs)
{
  // value() returns the storage_type of the type, so here it calls
  // standard C operator '+'.
  return lhs.value() + rhs.value();
}
```

But we also want to handle operations with builtins, such as int_u<8, strict> + **unsigned int**. A naive solution may be to add (1) :

```
template <class T1, class B2>
return_type
operator+(const rec_scalar<T1>& lhs,
          const B2& rhs)
{
  return lhs.value() + rhs;
}
```

and (2) :

```
template <class T1, class B2>
return_type
operator+(const B2& rhs,
          const rec_scalar<T1>& lhs)
{
  return lhs.value() + rhs;
}
```

But this code is ambiguous for the compiler, section A.2 gives a complete explanation of the problem, and gives several solutions. We chose a derivative of the last exposed one.

4.3.2 Implementation

Global operators are defined to get control over the operations being computed. The problem is how to find the good implementation for every operation. optraits is our solution. The following source clarify this idea :

```
template <class T1, class T2>
return_type operator+ (const T1& lhs, const T2& rhs)
{
  // Stuff to determine implementation type
  //...
  return optraits<implementation_type>::operator_plus (lhs, rhs);
}
```

Two unknown types are used in this code : return_type and implementation_type. Let us see how to determine them.

Determining return type

Once again, traits will help us. The idea is to define return type for each operator and for each type. Using this mechanism, we can implement type growing easily. This step is quite fastidious, even it there is several ways to simplify it. Here is an example of traits defined for int_s :

```
// Addition between two int_s
template <class nbits, class b1, class mbits, class b2>
struct operator_plus_traits<int_s<nbits, b1>, int_s<mbits, b2> >
{
  enum { commutative = 1 };
  // stuff to determine behavior from b1 and b2
  // ...
  typedef int_s<max<nbits,mbits>::ret + 1, behavior> return_type;
};

// ...
// Multiplication between an int_s and an int_u
template <class nbits, class b1, class mbits, class b2>
struct operator_times_traits<int_s<nbits, b1>, int_u<mbits, b2> >
{
  enum { commutative = 1 };
  // stuff to determine behavior from b1 and b2
  // ...
  typedef int_s<nbits+mbits+1, behavior> return_type;
};

//... other traits
```

Traits are defined for arithmetic, logical and comparison operators. You may have noticed the commutative value, which simplify the process implicitly defining define reverse traits when the return_type remains the same. You may also wonder how builtin types are handled this way, without defining traits for every one. These two features are possible thanks to a wrapper, deduce_from_traits. Indeed, the programmer should not use operator_xxx_traits directly. Here is the main principle of deduce_from_traits, in pseudo C++ :

```
#define NON_BUILTIN(T) // give non builtin equivalent type of T
                       // For example, NON_BUILTIN(unsigned char) returns
                       // int_u<8, strict>.

// ...

template <template<class, class> traits, class T1, class T2>
struct deduce_from_traits
{
  meta_if <traits<T1, T2> is defined>
    {
      typedef traits<T1, T2>::return_type return_type;
    }
  meta_else_if <traits<T2, T1> is defined
               and traits<T2, T1>::commutative is true>
    {
      typedef traits<T2, T1>::return_type return_type;
    }
  meta_else_if <traits<NON_BUILTIN(T1), NON_BUILTIN(T2)> is defined>
    {
      typedef traits<NON_BUILTIN(T1), NON_BUILTIN(T2)>::return_type return_type;
    }
  meta_else_if <traits<NON_BUILTIN(T2), NON_BUILTIN(T1)> is defined
               and traits<NON_BUILTIN(T2), NON_BUILTIN(T1)>::commutative is true>
    {
      typedef traits<NON_BUILTIN(T2), NON_BUILTIN(T1)>::return_type return_type;
    }
};
```

This dispatching is entirely static. Now global operators can use :

```
template <class T1, class T2>
deduce_from_traits<operator_plus_traits, T1, T2>::return_type
operator+ (const T1& lhs, const T2& rhs)
{
  // ...
}
```

Now remains one unknown type, implementation_type.

Determining implementation type

The answer is, guess what : traits. The idea is to insert implementation_type into all operator_xxx_traits. This lets maximal flexibility. Now we have the final code of global operators :

```
template <class T1, class T2>
deduce_from_traits<operator_plus_traits, T1, T2>::return_type
operator+ (const T1& lhs, const T2& rhs)
{
  typedef deduce_from_traits<operator_plus_traits, T1, T2>::impl_type impl_type;
  return optraits<impl_type>::operator_plus (lhs, rhs);
}
```

Final implementation of the operator

At last, in the optraits hierarchy, we can implement operator_xxx methods. All scalars operators are defined in optraits_scalar class, from which all scalar types inherit. Thus, the code is written only once :

```
template <class Self>
struct optraits_scalar
{
  template <class T1, class T2>
  deduce_from_traits<operator_plus_traits, T1, T2>
  operator_plus (const T1& lhs, const T2& rhs)
  {
    // ...
  }
};

template <unsigned nbits, class behavior>
struct optraits<int_u<nbits, behavior> >
  : public optraits_scalar<int_u<nbits, behavior> >
{
  // operator_plus is inherited
  // However, you may rewrite it, or add specializations.
}
```

The operator_plus method is implemented using section A.2 third solution principles.

4.4 Safety checks

We have the control on every operations, so it is easy to add checks in all constructors and after every calculus to ensure the result is valid. The big advantage we have is that thanks to strong typing and type minimal growing, we can generally avoid dynamic checks.

4.5 Decorators

Orthogonality of the type system will help us integrates decorators. Let us take range as an example to explain the implementation of decorators.

range will be inserted into the data hierarchy as a scalar. We cannot make it derivate from its argument (at least not directly) since we would like to decorate also builtin type.

The implementation of range just have to inherit the implementation of the decorated type, overriding min() and max :

```
template <class T, class interval, class behavior>
struct optraits<range<T, interval, behavior> > : public optraits<T>
{
  typedef interval::storage_type interval_type;

  static interval_type min()
  { return interval::min(); }

  static interval_type max()
  { return interval::max(); }

  // ...
};
```

Now range have a complete implementation, but what about traits needed for operators ? There is a way to specify in typetraits the type to use to resolve traits searches. This type is then used by deduce_from_traits to return the good result.

Chapter 5

Conclusion

5.1 Results

We have a working set of data types, already used in Olena 0.6. The current implementation works quite fine and satisfy most of our objectives :

- Secure data types, operations and assignments are always verified.
- Fast checks thanks to very strong typing.
- Good transparent and complete builtins interactions.
- Support for generic algorithms.
- Basic support for decorators.

However, several limitations darken these results.

5.2 Limitations

As this report may have given the impression, the code is rather complex and unusual. Heavy meta programming and static stuff certainly obfuscate things. Debugging is a real headache, and error messages provided by compilers are generally hard to understand. Furthermore, compilation time are dramatically increased, as there are many template instantiations and optimizations to perform.

5.3 Future work

5.3.1 Decorators

Decorators are not well integrated in the hierarchy. This is really annoyong for several reasons :

- A general decorator will be on top of the hierarchy, as we don't know where to put it. This prevents it to pass directly (without casts) through algorithms waiting for the decorated type.
- If there is particular methods in the data hierarchy for a type, the decrator won't get it.

These problems are being corrected, and decorators will inherit from the decorated type, or from the closest type if we are decorating builtin types.

5.3.2 Algebraic properties

Sometimes programs or algorithms works on particular algebraic types, such as group or monoid. Using conditional inheritance [5], it is possible to automatically inherit from the good structure, just describing the features of the type, such as having a relational order, a neutral element, ... This feature would be especially useful for image processing.

Chapter 6

Bibliography

[1] Matlab. http://www.mathworks.com.

[2] Octave. http://www.octave.org.

[3] Olena: A generic image processing library. http://www.lrde.epita.fr.

[4] Vigra: Generic computing for computer vision. http://kogs-www.informatik.uni-hamburg.de/ koethe/vigra/.

[5] David Lesage. Generic morphers. *Technical report, LRDE*, 2002.

[6] T. Veldhuizen. Techniques for scientific c++, 2000.

Appendix A

C++ technical points

A.1 Determinining parent type

It is possible in C++ to statically check if a type1 "is a" type2. This means type1 inherits from type2.

For example, in a function template taking two parameters, it is possible to ensure T1 inherits from T2 :

```
template <class T1, class T2>
void foo(T1 lhs, T2 rhs)
{
  is_a (T1, T2)::ensure();

  // ...
}
```

This also works with meta template parameters :

```
template <class T>
struct foo {};

// ...

template <class T1, class T2>
void bar(T1 lhs)
{
  // will abort compilation is T1 does not inherits from foo<X>
  is_a (T1, foo)::ensure();

  // ...
}
```

Technical note : is_a is implemented as a macro using several C++ tricks, but the main principle is based on C++ function overloading, consider the following code :

```
  struct A {};
  A* makeA();

  struct B : public A {};
  B* makeB();

  struct C {};
```

```
C* makeC();

typedef char _yes;
typedef int _no;

_yes foo(A*) {}
_no  foo(...) {}

int main() {
  sizeof( foo(makeA()) ) == sizeof( _yes ); // true, choose foo(A*)
  sizeof( foo(makeB()) ) == sizeof( _yes ); // true, choose foo(A*)
  sizeof( foo(makeC()) ) == sizeof( _yes ); // false, choose foo(...)
}
```

A.2 Overloading

A.2.1 Problems

Let's consider the following example :

We have a static hierarchy myValue, in which we find myInt, myFloat, etc ... We get the real (in sense of builtin) value of any of these types using the value() method : myType.value(). We want an operator+ beetween all these types :

We get (1) :

```
returnType operator+(const myValue<T>& lhs, const myValue<T>& rhs)
{
  return lhs.value() + rhs.value();
}
```

Now we want interactivity with builtins types, as they have no hierarchy we must define (2) :

```
template <class T1, class T2>
returnType operator+(const myValue<T1>& lhs, const T2& rhs)
{
  return lhs.value() + rhs;
}
```

and (3)

```
template <class T1, class T2>
returnType operator+(const T1& lhs, const myValue<T2>& rhs)
{
  return lhs.value() + rhs;
}
```

If you try to compile this code doing myInt() + myFloat() the compiler says that your call is terribly ambiguous. Indeed, to allow builtins types we said template <class T>, and as we cannot define any restrictions on T, it includes myInt or myFloat. You may think : "but i have defined an operator for myValue + myValue, he should call this one !", but in C++, overloading resolution finds easier to consider myInt() or myFloat() as a T than going through the hierarchy, so the two candidates are (2) and (3), and they have the same priority, no choice is possible.

A.2.2 Solution 1

Define the carthesian product of operators : operator+(myInt, myFloat), operator+(myInt, unsigned int), etc ... Quite tedious if you have many types !

A.2.3 Solution 2

Using meta programmation, the is_a tool described in section A.1, and a main generic interface (if_ stands for static if) :

```
template <class T1, class T2>
returnType operator+(const T1& lhs, const T2& rhs)
{
  if_ is_a(T1, myValue)
  then
      if_ is_a(T2, myValue)
      then "Call operator_plus_value_value"
      else "Call operator_plus_value_builtin"
  else
      if_ is_a(T2, myValue)
      then "Call operator_plus_notvalue_value"
      else "Call operator_plus_notvalue_notvalue"
}
```

Now we just have to define the 4 operator_plus_* and it's done. The first problem is compilation time, and it can be tedious if we have to choose beetween many categories of types (many levels of if_).

A.2.4 Solution 3

Let's consider a class anyClass defined as above :

```
template <class T>
struct anyClass
{
  anyClass(const T& t) : _target(t) {}
  T self() { return _target; }
  const T& self() const { return _target; }
  const T& _target;
};
```

anyClass can be constructed from everything, so that the cast is always possible to anyClass. Now consider the following code :

```
  template <class T1, class T2>
  returnType operator+(const myValue<T1>& lhs, const anyClass<T2>& rhs)
  {
    return lhs.value() + rhs.self();
  }
```

This operator will be chosen only if rhs is not a myValue, because the compiler prefers a travel into the hierarchy to a cast. In fact we are constraining the T, telling the compiler to choose this function is nothing else is possible. But it's not terminated, the compiler will never call this operator, as it cannot guess the T2 for anyClass which permit the cast. So we need to tell it that he should try with the type itself (for example, it the parameter is unsigned, it should try anyClass<unsigned>). That's why we need an interface with explicit instantiation.

```
returnType operator+(const T1& lhs , const T2& rhs ) { return
  operator_plus_impl<T1, T2>(lhs , rhs ); } then we can define

returnType operator_plus_impl(myValue<T1>, myValue<T2>);
returnType operator_plus_impl(myValue<T1>, anyClass<T2>);
returnType operator_plus_impl(anyClass<T1>, myValue<T2>);
```

Conclusion The 2 methods which works quite fine need a global operator, but it is not such a problem if we use namespace, as it will be explained in next section. Moreover, thanks to the compiler optimisations, there is no runtime cost.

A.3 Global operators - Namespace resolution

A.3.1 Typical problem

Considering the following code :

```
// Global operator
template <class T1, class T2>
void operator+ (const T1&, const T2&)
{
  // ...
}

struct A {};
struct B : public A {};

void operator+(const A&, const A&)
{
  // ...
}

int main()
{
  B() + B(); // calls operator+(T1, T2)
  // ...
}
```

Our global operator has superseded the operator+(A, A), this means using our data types operators will intercept user defined operators. Moreover, it will interfere with STL stream operators and avoid using standard types such as std::string.

A.3.2 Koenig lookup

Here is an example of Koenig lookup :

```
namespace foo {

  template <class T1, class T2>
  void operator+ (const T1&, const T2&)
  {
    // ...
  }
  struct myInt
  {
    // ...
  };

}

int main()
{
  foo::myInt(5) + foo::myFloat(6); // ok it finds operator+(T1, T2)
  // ...
}
```

This code shows the ability of C++ to search operators in the namespace of the arguments, here when using plus with foo::myInt, it searches the possible operators in the namespace of myInt.

A.3.3 Solution

So using Koening lookup we can hide global operators and types in a namespace safely, it will not supersede users operators. This has a considerable drawback, it obliges users to call types by their absolute names (ie with namespace), because an "using namespace foo" in the above code would break the separation and make the global operator visible to the world.

A good solution is to defines only types in a "public" namespace, designed for using namespace directives :

```
namespace foo {

  template <class T1, class T2>
  void operator+ (const T1&, const T2&)
  {
    // ...
  }

  struct myInt
  {
    // ...
  };

}

namespace foo_public {

  using foo::myInt;

}

struct A {};
struct B : public A {};

void operator+(const A&, const A&)
{
  // ...
}

using namespace foo_public;
```

```
int main()
{
  myInt(5) + myFloat(6); // ok it finds operator+(T1, T2)
  B() + B(); // ok it calls operator+(A, A)
  // ...
}
```