Theory of objects and application to the C++ language

Ignacy Gawędzki

Technical Report n^o0219 - October 2002

Many concepts described in the OOP literature are used in generic programming paradigms. The C++ language, though ubiquitous in industrial applications, does not natively provide the constructs to express these concepts naturally and forces programmers to use heavy template meta-programming techniques. While this permits the application of these concepts, it also renders the code very difficult to maintain.

We present an effort to augment a subset of the C++ language by adding natural support for simple constructs that enable the application of the wanted concepts. Besides, we establish simple rewriting rules to convert this extended subset to standard heavily-templated C++, implementable using code transforming tools.

De nombreux concepts décrits dans la littérature POO sont utilisés dans les paradigmes de programmation générique. Bien qu'il soit un langage répandu dans l'industrie, le C++ ne permet pas d'exprimer ces concepts naturellement et oblige le programmeur à utiliser des techniques de méta-programmation par utilisation lourde de templates. Ces techniques permettent l'application de ces concepts mais rendent le code difficile à maintenir.

Nous présentons une approche qui consiste à augmenter un sous-ensemble du C++ en ajoutant le support syntaxique naturel permettant d'exprimer les concepts voulus. Par ailleurs, nous établissons un ensemble de règles simples de conversion vers du C++ standard, en vue de les implémenter à l'aide d'outils de transformation de programmes.

Keywords
Syntactic sugar, 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

The design of a programming language can be a lengthy process, depending on the goals one wants to achieve. It is clear that an efficient compiler implementation precedes widespread use of a language and is time consuming as it requires maturing through testing and optimization. Meanwhile, language theorists go forward in formalizing new concepts that are very attractive for software engineers.

In the field of software engineering for scientific computing, the application of these new concepts is desirable, as we shall see later, but the design of a new language from the ground up out of reach or simply pointless for the reasons given above.

1.1 The context

First of all, we set ourselves in the realm of **static generic programming in C++** which is very much desirable in the field of scientific computing.

Generic programming The reasons for the preference of generic programming are simple and pretty much the same for any field of software engineering: code **factorization** and **reusability**. Generic code is written at a higher level of abstraction in respect to some input data as some data types, structures and/or parameters. Therefore, the bet is that the process of specialization of the generic code for some given input data will be simpler and faster — in fact in most of the cases it can be automated — than the writing of the whole code anew. It derives from this point that the simple fact of abstracting some algorithm is a process of code factorization: the same code can be used for various data inputs. It is then clear that code factorization allows simpler code reuse, as the specialization can be applied later and in other contexts.

Static programming The reasons for the preference of static programming are more specifically related to the field of scientific computing, which often implies huge amounts of data to process. The primary goal is then to produce **efficient code**, in order to perform the huge computation tasks as fast as possible. Compilers are required to perform specializations given the static information about input data and then optimize the code as much as possible, in order to achieve code efficiency comparable to dedicated code. The other important aspect of static programming is **security**. The compiler is given all the typing information and guarantees that if the code compiles, it is type-sound and no dynamic type error can ever happen. This has the interesting implication that the compiler can thus remove any dynamic typing checks from the resulting code, rendering the execution even more efficient.

The C++ language It has the advantage of being a widespread **industrial class** language. The compilers benefit from the experience acquired for C compilers and thus can produce efficient code. In addition, it happens that it is possible to write static generic programs in it, so it is an interesting choice for scientific applications.

1.2 Implications

The use of C++ presents some annoying drawbacks: writing code in C++ using static genericity paradigms implies heavy use of **template meta-programming techniques**.

Lengthy compilations Compilation times tend to be very long. This is caused by the requirement of the compiler to execute the meta-programming clauses, which means lots of template instantiations. In other words, it take a long time for the compiler to specialize and optimize the generic code we have written.

Cryptic error messages Error messages become cryptic, because errors can have consequences buried deeply into template instantiation.

Cryptic code Worse point of all, the code that one has to write in order to apply static genericity paradigms is very complicated, making it more difficult to maintain and leading more easily to errors.

1.3 Existing answers

The problem of lengthy compilations is not really a concern in scientific computing, since we can ever assume that computing time is longer by several orders of magnitude. In other words, we can afford lengthy compilations.

The problem of cryptic error messages is not our concern in this report, but is addressed in some efforts (see [7] for example). Addtitional static checks allows for errors to be detected very early in the instantiation process, hence shorter error messages that can even be customized to some extent.

Throughout this report, we focus on the last point, as we present an effort to "augment" a subset of the C++ language, by adding support for new constructions, in order to let us write code that is more *readable*.

1.4 Our goals

Observations First, we obviously will not change the C++ standard, as any slight change proposal takes a long time to make it into the standard and our concerns are unfortunately not the standard makers' priority. Second, the concepts of object-oriented programming we want to use are simple, as we shall see in chapter 2. Lastly, we have powerful tools for the application of program transformation (see [3]).

Decision It becomes clear that we want to transform some sort of "augmented" C++ into standard C++ as described in [1]. We thus benefit from mature C++ compilers and let us switch them as they change or evolve.

1.5 Report outline

We first present some interesting concepts presented throughout the literature on objects theory. Then we present the C++ features as they have been provided by its designers, the twisted way in which we can use them and the way we would like them to be. Lastly, we present the way in which we can transform this "augmented" C++ into standard C++.

Chapter 2

Typing in objects theory

2.1 Typing basics

Typing in object oriented languages has been a topic of great interest during the last decade.

The type checking rules of a given language define the way code soundness has to be checked to prevent obviously bad instructions from being evaluated. For an extensive introduction to typing theory, please refer to [6] and [2].

In its simplest form, a **type** can be seen as the set of all values that have that type.

$$A$$

For example, Int can be defined to be the set of integers, Float can be defined as the set of floating-point values.

To say that a particular variable a belongs to a particular set, i.e. that it is of type A, we can simply write:

$$a : A$$

The type of a tuple, similarly as in set theory, is a product of several types.

$$(a_1, a_2, \dots, a_n) : A_1 \times A_2 \times \cdots \times A_n$$

For example, consider the couples where the first member is an integer and the second is a floating-point value. The type of the couple is simply $\mathit{Int} \times \mathit{Float}$.

Record types are similar to tuples, as they are a type composition of the members' types, up to the fact that order does not matter (not from a typing standpoint).

$$\{a_1, a_2, \dots, a_n\} : \{A_1, A_2, \dots, A_n\}$$

Formally, functions are also variables, but their type includes the type of the arguments — all the arguments can be seen as a tuple — and the type of the returned values.

$$f : A_1 \times A_2 \times \cdots \times A_n \rightarrow B$$

2.2 Subsumption and subtyping

To apply generic programming paradigms, we want support for **polymorphism**, i.e. functions that can be applied to values of several different types in order to achieve reusability. There are several forms of polymorphism that can coexist in a given language, as shown by Cardelli and Wegner in [6]. Its simplest form is **inclusion polymorphism** which is based on **subsumption**, i.e. the ability of values of some type to be seen as values of another type. Consider, for example, a function that takes one argument, an integer encoded on 16 bits and returns a boolean (that function could for example check if the value is positive, or if the number is a prime number). From a practical standpoint, the difference between integers encoded on 16 bits and integers encoded on 8 bits is only the range of values they can hold, but the way other operations are performed does not change. Suppose we want to apply the function on values encoded on 8 bits. We know that in terms of range, any value encoded on 8 bits also fits in a 16 bits variable. Therefore, we have the guaranty that any value encoded on 8 bits can be encoded on 16 bits without loss. Then we say that 8 bits values subsume 16 bits values.

If we consider types instead of values, we are talking about **subtyping**, which is expressed in the following manner:

$$A <: B \iff (a : A \implies s : B), \forall a$$

We say that "A is a subtype of B".

Let $Short$ be the type of integers encoded on 8 bits and $Long$ the type of integers encoded on 16 bits. Another way of saying that values of type $Short$ subsume values of type $Long$ is to say that $Short$ is a subtype of $Long$.

If we consider tuples, subtyping for composite types is defined as follows:

$$A \times A' <: B \times B' \iff A <: B \text{ or } A' <: B'$$

Here, we see that the $\times$ operator does not change the direction of the subtyping relation for both of its members. It is said to be **covariant** in respect to the $<:$ relation for both members, because their types must change *along* with the composite type.

Similarly, we define subtyping for record types:

$$\{A_1, \ldots, A_n\} <: \{A'_1, \ldots, A'_m\} \iff m < n \text{ and } \forall i_{1 \leq i \leq m}, A_i <: A'_i$$

Note that a record type's subtype can have more members, as long as it provides at least those of the original type. This is illustrated by figure 2.1.

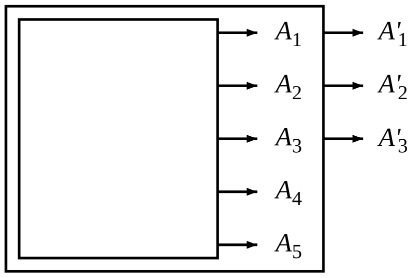

Figure 2.1: Subsumption of record types

The subtyping relation is also defined for function types:

$$S \to E <: S' \to E' \iff S' <: S \text{ and } E <: E'$$

The $\to$ is covariant in for the return type (the second member) but not for the arguments' type. It is said to be **contravariant** for its arguments' type, because it is required to change in the opposite direction than the function type.

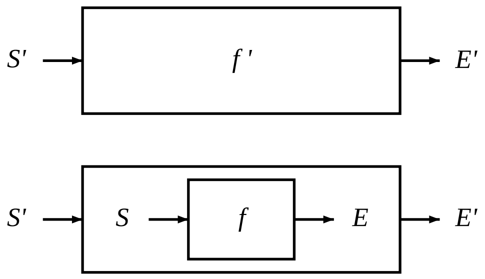

Figure 2.2: Subsumption of function types: we want $f <: f'$.

This behavior may seem counterintuitive at first glance but can be explained easily. If we bear in mind that subtyping comes from the need for subsumption, we must remember that any value of a subtype of the original type must fit where a value of the original type is expected. The covariance of the return type is due to the fact that whatever the function returns must fit into the assumed return type (i.e. the return member of the original type). The contravariance of the arguments' type is simply due to the fact that it must hold at least all the possible values that fit into the original arguments' type (see figure 2.2).

If we look at some variable that can be updated (i.e. not only read but also written to), we see that its type is in fact **invariant**, i.e. it has no non-trivial[1] subtypes. This comes from the fact that the value can be read, implying covariance and written to, implying contravariance, hence invariance (see figure 2.3).

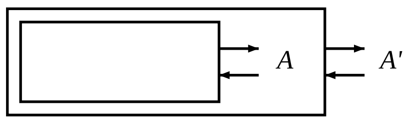

Figure 2.3: Subsumption of mutable variable types

2.3 Object typing

Now let's go a bit further and see how we can consider objects' types. An object's type is what is seen from the outside, i.e. it must take into account only *public* members. An object's type is in fact a record type of the public attributes and methods:

$$o : \{A_1, \ldots, A_n, M_1, \ldots, M_m\}$$

Then, object type subtyping is exactly the same as record type subtyping.

2.4 Objects' types and objects' classes

So far, we have talked about values/objects and types, but nothing about classes. It is time to present some distinctions that are used in the object typing literature. In OOP, there are two kinds of languages: **object-oriented** and **class-oriented**.

In the first kind, there are constructions that allow *ad hoc* object construction. One can either create a wholly new object by composition of its attributes and methods or create a new object by taking an existing object and by changing or adding some attribute(s) or method(s).

In the second kind, objects are created by **class instantiation**. One must be careful not to mix up types and classes. These are utterly different concepts. As we have said before, a object's type is its external "appearance" (sometimes called **signature** or **interface**), whereas its class is a description of its structure, which comprises not only public but also private type information and initial values for all the members.

2.5 Subclassing (inheritance)

Having introduced classes, we can now present another important concept: **class inheritance** (denoted $<_C$). It is the ability to define new classes by reusing some existing class, in a way similar to object creation by reuse in object-based languages. Therefore, one must remember that **inheritance does not mean subtyping** and that **inheritance does absolutely not imply subtyping**. It happens that some languages (to which C++ belongs) require subclasses to produce objects in subtyping relation and thus bringing only confusion to programmers.

2.6 Some practical examples

To introduce some additional concepts, we shall see some examples that exhibit problems solved by these concepts. The following code examples are written in a pseudo language, bearing resemblances to pseudo-languages used in the literature (the example is taken from [4]).

Suppose we want to define a class $Point$ used to produce objects of type $PointType$.

[1]Trivial subtypes exist for record types: they only differ by the order of the members.

```
   class Point
2    var
       x := 0 : Int;
4      y := 0 : Int
     methods
6      function eq(rhs : PointType) : Bool
         begin
8          return x = rhs.x & y = rhs.y
         end
10 end class;
```

This class is interesting because it has a **binary method**, i.e. a method that takes an argument of the same type as objects generated by the currently defined class.

As we see in the code above, the dereferences on line 8 are legal, because `rhs` being of the type of the objects generated by the current class is guaranteed to have the `x` and `y` attributes.

Now suppose we want to reuse that code in order to define another class called $ColorPoint$, supposing that we are allowed to change the method's type. This is very convenient for classes with binary method as we would like to assume that the argument provides the same attributes and methods as the currently defined class.

```
   class ColorPoint inherits Point
12   var
       c := black : ColorType
14   methods
       function eq(rhs : ColorPointType) : Bool
16       begin
           return x = rhs.x & y = rhs.y & c = rhs.c
18       end
   end class;
```

By writing, on line 11, that $ColorPoint$ inherits from $Point$, we say that we want the class $ColorPoint$ to contain all the attributes and methods of $Point$. But here, we redefine the `eq()` method to take into account the additional attribute.

On line 17, we can see that the dereferences are correct, provided that `rhs` has the type $ColorPointType$ as requested (we have the `x`, `y` and `c` attributes).

Although the two classes are by definition in the subclass relation,

$$ColorPoint <_C Point$$

the types of the generated objects are not in the subtype relation:

$$\begin{aligned} PointType &= \{Int, Int, PointType \to Bool\} \\ ColorPointType &= \{Int, Int, ColorType, ColorPointType \to Bool\} \end{aligned}$$

$$\begin{aligned} PointType &\not<: ColorPointType \\ ColorPointType &\not<: PointType \end{aligned}$$

The first clause is evident, because $PointType$ has less attributes than $ColorPointType$. Then if $ColorPointType$ where a subtype of $PointType$, we would have to have the following relation:

$$ColorPointType \to Bool <: PointType \to Bool$$

Then by application of the subtyping relation on functions for the method `eq()`, we see that we would have to have:

$$\mathit{PointType} <: \mathit{ColorPointType}$$

Which we have proved false, hence the second clause.

Then, we are forbidden to write the following code:

```
20 var
     p1 : PointType;
22   p2 : ColorPointType
   begin
24   p2.eq(p1);
     p1.eq(p2)
26 end
```

The method calls on lines 24 and 25 are not type-sound, because $\mathit{PointType} \not<: \mathit{ColorPointType}$ and $\mathit{ColorPointType} \not<: \mathit{PointType}$ respectively. This is good if we do not plan to compare *Point*s with *ColorPoint*s.

Now suppose we would like to reuse the method defined in the parent class in the body of the redefined method in the child class without having to re-write it entirely. We could use the **`super`** keyword to access the parent class's method `eq()`. But we would not be allowed to call it within the body of the redefined method with the `rhs` argument. This is simply because the `eq()` method of the parent class takes an argument of type *PointType* and we want to apply it on an argument of type *ColorPointType*. This fact is pretty annoying since we know that the idea is sensible, as all the required attributes are present in the argument for a successful execution of the parent class's method.

2.7 The relation of matching

To allow easy handling of binary methods, we have first to introduce a special type called *SelfType* (dubbed *MyType* by Bruce in [4]). It stands for the type of the objects generated by the currently defined class. Now we simply replace the type of the method's argument with *SelfType*.

```
   class Point
2    var
       x := 0 : Int;
4      y := 0 : Int;
     methods
6      function eq(rhs : SelfType) : Bool
         begin
8          return x = rhs.x & y = rhs.y
         end
10 end class;
   class ColorPoint inherits Point
12   var
       c := black : ColorType
14   methods
       function eq(rhs : SelfType) : Bool
16       begin
           return super.eq(rhs) & c = rhs.c
18       end
   end class;
```

For the purpose of typechecking, *SelfType* is valid only in method definitions and nowhere else. It is supposed to be an unconstrained free type variable.

Now, the code on lines 8 and 17 is valid.

We can now introduce the relation of **matching** (denoted $<_\#$): relation of subtyping with the supposition that $\mathit{SelfType}$ matches the type of the objects generated by the currently defined class.

Thus we have a straightforward implication, if we suppose we cannot change methods' types in subclasses.

$$C <_C C' \implies \mathit{ObjectType}(C) <_\# \mathit{ObjectType}(C')$$

Another implication is of course that subtyping implies matching:

$$A <: A' \implies A <_\# A'$$

Thanks to the matching relation, the above code is correctly typed and prevents us from comparing Points with $\mathit{ColorPoint}$s.

On the other hand, we are stuck here if we want polymorphism over Points and $\mathit{ColorPoint}$s. Inclusion polymorphism is not enough.

2.8 Match-bounded parametric polymorphism

To solve this, we have **parametric polymorphism**. It is a way of explicitly abstracting some parameters such as type or integer variables in a function or class definition in order to express identity relations on the parameters that the use of inclusion polymorphism does not allow. The C++ language allows the use of parametric polymorphism with the use of templates.

Without parametric polymorphism, we have been required to write some code twice, one for $\mathit{PointType}$ and one for $\mathit{ColorPointType}$.

```
function foo(p1 : PointType, p2 : PointType) : Bool
  begin
    return p1.eq(p2)
  end
function foo(p1 : ColorPointType, p2 : ColorPointType) : Bool
  begin
    return p1.eq(p2)
  end
```

This seems a bit silly, since it would only require the compiler to know that `p1` and `p2` are of the same type to factorize this into one polymorphic function.

```
function foo(PType, p1 : PType, p2 : PType) : Bool
  begin
    return p1.eq(p2)
  end
```

If a function has two parameters of the same type, inclusion polymorphism allows the application of the function on two objects whose types may be subtypes of the required type *independently*. One cannot express, for example, that the two exact types have to be the same.

Moreover, parametric polymorphism allows polymorphism on types that are in no relation at all. In the theoretical case, this has the bad property of not being typecheckable without prior specialization of the code for the specific types it is applied on. This is the way C++ compilers do to typecheck templates. This is due to the fact that no assumption can be made about the parameters, since they are presented as completely abstract types.

To overcome this problem, theorists have introduced a way to constrain the parameters in order to allow typechecking to be performed on the partially abstracted code: **bounded parametric polymorphism**. It is done by simply bounding the parameter, constraining it to be a subtype of a given known type. This way, the parameter is guaranteed to provide at least methods and attributes of the bound.

Unfortunately, this does not solve in turn the case where we want to abstract over types of objects with binary methods. This requires us to use the matching relation to express the bound, hence **match-bounded parametric polymorphism**.

Now we can rewrite the `foo()` function to be polymorphic on types that match *PointType*:

```
function foo(PType <# PointType, p1 : PType, p2 : PType) : Bool
  begin
    return p1.eq(p2)
  end
```

With this code, we prohibit the use of `foo()` with two parameters of different type or of type that does not match *PointType*.

Another example often cited in the literature is the following:

```
class Circle(CenterType <# PointType, origpoint : CenterType)
  var
    center := origpoint : CenterType;
    radius := 1 : Int
  methods
    function getcenter() : CenterType
      begin
        return center
      end;
    procedure setcenter(newcenter : CenterType)
      begin
        center := newcenter
      end
end class;
class ColorCircle(CenterType <# ColorPointType, origpoint : CenterType)
    inherits Circle(CenterType, origpoint)
  var
    color := red : ColorType
  methods
    function getcolor() : ColorType
      begin
        return color
      end;
    procedure setcolor(newcolor : ColorType)
      begin
        color := newcolor
      end
end class;
```

Here, the interesting thing is that we parameterize classes to allow clever changes in the types of attributes and methods. This way, we do not only allow for easy definition of binary methods, but also for other methods and attributes which type change along inheritance. The *ColorCircle* class inherits all the attributes and methods of *Circle* but binds the *CenterType* parameter tighter.

2.9 F-bounded parametric polymorphism

F-bounded parametric polymorphism, introduced for OO programming by Canning et al. in [5], is another way of solving the problem of polymorphism over types that exhibit some interesting aspects. This time, the bound is a type function applied on the parameter itself. It allows to express structural constrains on the parameter without intrusion into the classes' definitions.

A good example is the following code:

```
ftype Movable(Type)
  begin
    move : Real * Real -> Type
  end;
procedure translate(PType <: Movable(PType), point : PType, x : Real, y : Real)
  begin
    point.move(x, y)
  end;
```

We first define a function over types that produces a type that provides a method `move()` which takes two reals as arguments and returns the type itself. It allows us to then define a `translate` function that can only assume that the `point` parameter has the `move()` method that has the type $Real \times Real \to PType$.

2.10 Virtual types

The introduction of *SelfType* allowed to change binary methods' argument type covariantly while ensuring the good usage of the methods. If we extend this trick to any type, we can then change any argument's type covariantly in a secure way. This is where **virtual types** come in handy (see [10], for more information on static virtual types).

A virtual type can be seen as a partial type definition inside a class. Nevertheless it has to be completely defined in the exact class, otherwise there is no way to produce concrete objects.

The typical example used to illustrate this is the following code. We first define a *Food* hierarchy:

```
   class Food
 2   ...
   end class;
 4 class Grass inherits Food
     ...
 6 end class;
   class Meat inherits Food
 8   ...
   end class;
```

Then we define the *Animal* hierarchy that contains (see line 12) the definition of a virtual type *FType* that is initially match-bounded to *FoodType*, the type of objects generated by the *Food* class.

Then in the *Cow* class, *FType* is definitely set to *GrassType* (see line 19), the type of objects generated by the *Grass* class.

```
10 class Animal
     types
12     FType <# FoodType
     methods
14     procedure eat(f : FType);
       ...
16 end class;
   class Cow inherits Animal
18   types
       FType = GrassType;
20   methods
```

```
      ...
22 end class;
```

Then we write a procedure that takes some *Animal* and applies its `eat()` method on an object of its own *FType*.

```
   procedure keeper(a : Animal)
24   begin
       a.eat(new a.FType)
26   end
```

The code on line 25 is type-sound because the method `eat()` wants one argument of type *FType* and we give it exactly this. Whatever be the concrete type of `a`, it is guaranteed that `new a.FType` produces an object which type matches *FType*.

If we try to write incorrect code, it is not type-sound.

```
   procedure badfarmer(c : Cow)
28   begin
       c.eat(new Meat)
30   end
```

This is caused by the fact that the type of objects generated by the *Meat* class does not match `c.FType`. For typechecking purposes, we can assume, as for *SelfType* that a virtual type is a free unconstrained variable.

Chapter 3

Typing in C++

Having seen all these interesting theoretical concepts, let's see what is the position of the C++ language.

3.1 The C++ as it was designed

Class-based typing The typing of the C++ language is based on class definitions. That means that subsumption with inclusion polymorphism is based on labels and not structure. If one wants to be able to use one instance of a class where an instance of another class is required, she has to make the former inherit from the latter. Therefore, we can say that in C++, subtyping implies subclassing.

Covariant method return types Since 1998, the C++ standard allows covariant change to methods' return types in child classes. This allows narrowing of the return type that is in accordance with the subsumption property for objects generated by the classes.

Invariant method argument types A method's signature is not allowed to change in any way from the parent class to the child class.

No virtual types Only methods can be partially defined in classes, which gives birth to abstract methods and thus abstract classes. Any type used inside a class must be known, so partial type definitions are forbidden.

Static unbounded parametric polymorphism The templates were introduced to allow for parametric polymorphism on classes and functions. Unfortunately, parameters are unbound, hence the typechecking pass is done after template instantiation.

Limited expressiveness Even in case of simple requirements, these limitations force the programmer to write explicit dynamic type checks that slow the code down and do not catch static type errors at compile time, letting an exception be thrown at run time.

Let's look at a simple example:

```
class A {
  int _attr;
public:
  virtual bool eq(const A& rhs) const {
    return _attr == rhs._attr;
  }
  // ...
};
class B : public A {
  int _otherattr;
```

```
     public:
12     virtual bool eq(const A& rhs) const {
         B* rhsp;
14       if (!(rhsp = dynamic_cast<B*>(&rhs)))
           throw bad_typeid;
16       return A::eq(rhs) &&
                _otherattr == rhsp->_otherattr;
18     }
       // ...
20   };
```

As said before, the argument types are invariant, forcing us to define `B::eq()` as taking an argument of type **`const`** `A&`. But we would like to restrict the use of this method to arguments of exact type **`const`** `B&`, so we have to perform a dynamic type cast (line 14) in order to check the type conformance and dereference the `_otherattr` attribute. If the first condition does not hold, we have to throw an exception (line 15).

If we write the following code:

```
     int main()
22   {
       A     a;
24     B     b;
       b.eq(a);
26     return 0;
     }
```

the execution will fail unconditionally, throwing each time the **`bad_typeid`** exception.

This is really a shame because the compiler knows the exact type of `a` and `b` at compile time and could prevent such errors at compile time.

3.2 What we can do with it

First discovered by Erwin Unruh (see [11]), template meta-programs allow execution of arbitrary code at compile time (see [13] and [12]). Indeed, the template instantiation and specialization mechanisms turned out to provide a static programming level that is turing complete. Therefore, the C++ can be seen as a two-level language which allows **meta-programming**. So it is possible to make the compiler perform additional checking and optimization tasks at compile time.

Besides, template specialization comes in handy to define functions over types, called **traits**. In this way, we can store additional static type information useful for template meta-programs for type handling.

In the specific case of **closed-world compilation**, which is the case for static generic programming paradigms, every object's exact type is known statically and no object with unknown type can ever appear.

3.2.1 Recurring static hierarchies

One way of informing the compiler on objects' exact types is the use of **recurring static hierarchies**: each class is in fact a meta-class parameterized by its child class or the special class `Bottom`.

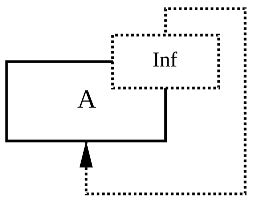

Figure 3.1: Recurring static hierarchies

This way, it is possible to recover the exact class of any object statically, by unrolling the hierarchy until the class is parameterized by `Bottom`, which indicates that the class is the exact class.

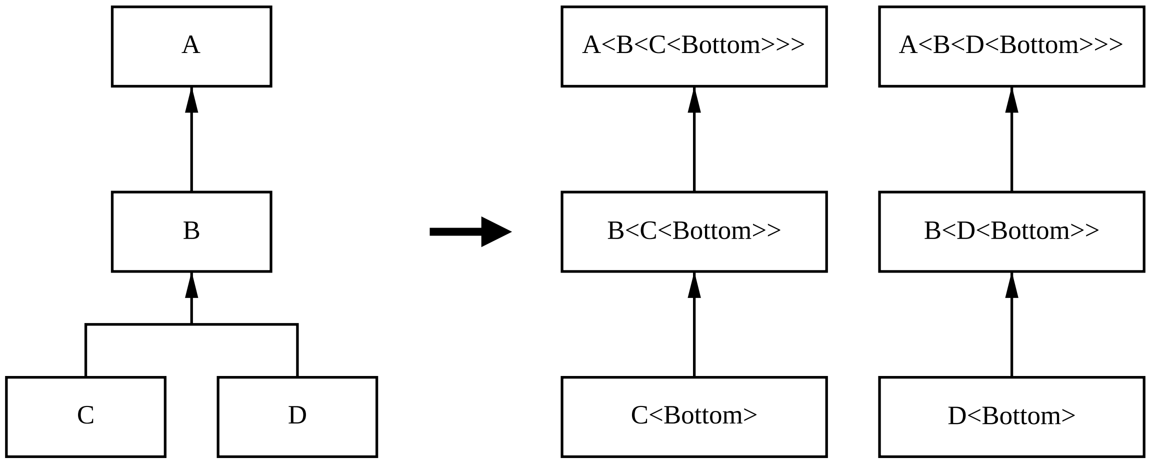

Figure 3.2: Unrolled hierarchies

Let's see a concrete example:

```
// definition of meta-programming tools
// ...
template <class> A_;
template <class Inf>
struct vt_trait< A_<Inf> > {
  typedef Inf   InferiorType;
};
template <class Inf = Bottom>
class A_ : public Top {
public:
  typedef A_<Inf>       SelfType;
// ...
};
typedef A_<>    A;
template <class> B_;
template <class Inf>
struct vt_trait< B_< A_<Inf> > > : public vt_trait< A_<Inf> > {
  typedef Inf   InferiorType;
};
template <class Inf = Bottom>
class B_ : public A_< B_<Inf> > {
public:
  typedef B_<Inf>       SelfType;
// ...
};
typedef B_<>    B;
// ...
```

This is the static skeleton of a very simple hierarchy, namely a base class A and a child class B.

3.2.2 Rewriting the *Point*/*ColorPoint* code

Let's see how the example of *Point* and *ColorPoint* looks like when written with this model:

```
    #define Dispatch(Meth) to_exact(this)->Meth ## _impl
 2  template <class Inf = Bottom>
    class Point_ : public Void< Point_<Inf> > {
 4  public:
      typedef Point_<Inf>   SelfType;
 6    bool eq(const SelfType& rhs) const {
        return Dispatch(eq)(to_exact(rhs));
 8    }
      bool eq_impl(const SelfType& rhs) const {
10      return x == rhs.x && y == rhs.y;
      }
12    // ...
    private:
14    int   x;
      int   y;
16  };
    template <class Inf = Bottom>
18  class ColorPoint_ : public Point_< ColorPoint_<Inf> > {
    public:
20    typedef ColorPoint_<Inf>      SelfType;
      typedef Point_<SelfType>      SuperType;
22    bool eq_impl(const SelfType& rhs) const {
        return SuperType::eq_impl(rhs) && c == rhs.c;
24    }
      // ...
26  private:
      Color c;
28  };
```

The definitions of the `to_exact()` function and the traits are omitted for the sake of clarity, see [8] for an extensive talk on the matter. The noteworthy aspects are that the base class implements a dispatching method that calls, on line 7, the method's implementation in the exact classes, defined on lines 9 and 22.

This code is strictly equivalent to the code presented in 2.7. We can check it with this simple code:

```
    int main()
30  {
      Point         p1, p2;
32    ColorPoint    cp1, cp2;

34    p1.eq(p2);
      p2.eq(p1);
36    cp1.eq(cp2);
      cp2.eq(cp1);
38    p1.eq(cp1); // error
      cp1.eq(p1); // error
40
      return 0;
42  }
```

The compiler does not allow code on lines 38 and 39 to compile and outputs the following errors:

```
In function 'int main()':
38: no matching function for call to 'Point_<Bottom>::eq (ColorPoint &)'
candidates are: bool Point_<Bottom>::eq(const Point_<Bottom> &) const
39: no matching function for call to 'ColorPoint_<Bottom>::eq (Point &)'
candidates are: bool Point_<ColorPoint_<Bottom> >::eq(const
Point_<ColorPoint_<Bottom> > &) const
```

Indeed, it is telling us that it can only compare *Point*s with *Point*s and *ColorPoint*s with *ColorPoint*s.

3.2.3 Using match-bounded parametric polymorphism

We can also use some kind of match-bounded parametric polymorphism.

```
   template <class T>
 2 bool foo(Point_<T>& p1, Point_<T>& p2)
   {
 4   return p1.eq(p2);
   }
```

We have expressed the requirement that `p1` and `p2` be of the same exact type.
Let's write some valid and invalid code:

```
 6 int main()
   {
 8   Point        p1, p2;
     ColorPoint   cp1, cp2;
10
     foo(p1, p2);
12   foo(cp1, cp2);
     foo(p1, cp1); // error
14   foo(cp1, p1); // error

16   return 0;
   }
```

The compiler prevents the errors on lines 13 and 14 and outputs the following errors:

```
In function 'int main()':
13: no matching function for call to 'foo (Point &, ColorPoint &)'
14: no matching function for call to 'foo (ColorPoint &, Point &)'
```

This works thanks to C++'s type inference rules for template parameters.

3.2.4 Using virtual types

We can take the example from section 2.10 and express virtual types by means of traits.

First the *Food* hierarchy:

```
template <class> class Food_;
template <class Inf>
struct vt_trait< Food_<Inf> > {
  typedef Inf           InferiorType;
};
template <class Inf = Bottom>
class Food_ : public Void< Food_<Inf> > {
public:
  typedef Food_<Inf>    SelfType;
  // ...
};
```

```
typedef Food_<> Food;
template <class> class Grass_;
template <class Inf>
struct vt_trait< Grass_<Inf> > :
    public vt_trait< Food_< Grass_<Inf> > > {
  typedef Inf           InferiorType;
};
template <class Inf = Bottom>
class Grass_ : public Food_< Grass_<Inf> > {
public:
  typedef Grass_<Inf>   SelfType;
  // ...
};
typedef Grass_<> Grass;
template <class> class Meat_;
template <class Inf>
struct vt_trait< Mean_<Inf> > :
    public vt_trait< Food_< Meat_<Inf> > > {
  typedef Inf           InferiorType;
};
template <class Inf = Bottom>
class Meat_ : public Food_< Meat_<Inf> > {
public:
  typedef Meat_<Inf>    SelfType;
  // ...
};
typedef Meat_<> Meat;
```

Then the *Animal* hierarchy.

```
template <class> class Animal_;
template <class Inf>
struct vt_trait< Animal_<Inf> > {
  typedef Inf           InferiorType;
  typedef Food          FoodType;
};
template <class Inf = Bottom>
class Animal_ : public Void< Animal_<Inf> > {
public:
  typedef Animal_<Inf>  SelfType;
  typedef vt(FoodType)  FType;
  void eat(const FType& f) const {
    Dispatch(eat)(to_exact(f));
  }
  // ...
};
typedef Animal_<> Animal;
template <class> class Cow_;
template <class Inf>
struct vt_trait< Cow_<Inf> > :
    public vt_trait< Animal_< Cow_<Inf> > > {
  typedef Inf           InferiorType;
  typedef Grass         FoodType;
};
template <class Inf = Bottom>
class Cow_ : public Animal_< Cow_<Inf> > {
   public:
66   typedef Cow_<Inf>     SelfType;
     typedef vt(FoodType)  FType;
68   void eat_impl(const FType& f) const {
       // ...
70   }
     // ...
72 };
   typedef Cow_<>  Cow;
```

Notice the additional entries in the `vt_trait<>` hierarchy, on lines 43 and 61.

Now some good and bad code.

```
74 template <class T>
   void keeper(const Animal_<T>& a)
76 {
     a.eat(*new vt_trait<Exact(a)>::FoodType);
78 }
   template <class T>
80 void badfarmer(const Cow_<T>& c)
   {
82   c.eat(*new Meat);
   }
84 int main()
   {
86   Cow c;
     keeper(c);
88   badfarmer(c);

90   return 0;
   }
```

The compiler complains about the bad code and tells us the following:

```
In function ‘void badfarmer<Bottom>(const Cow_<Bottom> &)’:
88:   instantiated from here
82: no matching function for call to ‘Cow_<Bottom>::eat (Meat_<Bottom> &) const’
50: candidates are: void Animal_<Cow_<Bottom> >::eat(const Grass_<Bottom> &) const
```

In fact, it is telling us that it does not know how to make a Cow eat $Meat$ and that it only can make it eat $Grass$.

3.3 How it could look like

We have seen that in fact many fancy theoretical concepts can be expressed in an obfuscated manner in standard C++ and that they actually are in some projects where static genericity is needed using C++.

The first thing we notice is that we use very obscure tricks and techniques to express concepts that are simple in essence. The second is that much of the added obscure code is very much the same for each class or function definition. Writing programs this way, we keep in mind the higher-level concepts and apply by hand the tricks and techniques to express them in standard C++. There is a strong feeling that this task could be partially automated: we would like to write programs in a clear, readable form that would be transformed into static C++.

Thanks to the work of Anisko and Tisserand (see [3] and [9]), we have the necessary tools to parse simple C++ code and to manipulate it (we are using the Stratego and XT package, see [16], [15] and [14] for more information). Moreover, these tools are very easily modifiable, so we can adapt them to process input code written in an "augmented" form of C++ that would exhibit natural constructs to express the wanted concepts. Then it would be pretty simple to output standard C++ code. We feel that the augmented form should be as close to standard C++ as possible, to let the transformation rules remain simple and most of the functional code be simply passed along.

3.3.1 Rewriting *Point*/*ColorPoint*

The C++ language could be augmented to include a **selftype** keyword that would have the meaning of *SelfType* and a **supertype** that would represent the parent class.

Then the *Point*/*ColorPoint* example would look like this:

```
class Point {
public:
  virtual bool eq(const selftype& rhs) const {
    return _x = rhs._x && _y == rhs._y;
  }
  // ...
private:
  int _x;
  int _y;
};
class ColorPoint {
public:
  virtual bool eq(const selftype& rhs) const {
    return supertype::eq(rhs) && _c == rhs._c;
  }
  // ...
private:
  Color _c;
};
```

3.3.2 Rewriting *Circle*/*ColorCircle*

Match-bound parametric polymorphism could be expressed by introducing the bound in the template parameter list.

```
template <class CenterType : Point>
class Circle {
public:
  const CenterType& getcenter() const {
    return _center;
  }
  void setcenter(CenterType& newcenter) {
    _center = newcenter;
  }
  // ...
private:
  CenterType _center;
  int        _radius;
};
template <class CenterType : ColorPoint>
class ColorCircle : public Circle<CenterType> {
public:
```

```
18    const ColorType& getcolor() const {
        return _color;
20    }
      void setcolor(ColorType& newcolor) {
22      _color = newcolor;
      }
24    // ...
    private:
26    ColorType _color;
    };
```

Notice the : construct in the template parameter list on lines 1 and 15.

3.3.3 Rewriting $Cow/\ldots$

Virtual types could be supported by the addition of the **virtual** qualifier to **typedef** lines inside classes.

```
    class Food {};
 2  class Grass : public Food {};
    class Meat : public Food {};
 4  class Animal {
    public:
 6    virtual typedef Food  FType;
      virtual void eat(FType& f) = 0;
 8  };
    class Cow : public Animal {
10  public:
      virtual typedef Grass FType;
12    virtual void eat(FType& f) {
        // ...
14    }
      // ...
16  };
    void keeper(Animal& a)
18  {
      a.eat(*new a.FType);
20  }
    void badfarmer(Cow& c)
22  {
      c.eat(*new Meat);
24  }
```

Notice that the instruction on line 19 requests an instantiation of the type `a.FType` which looks rather like a member of the object `a`. We do not want to write `a::FType` since `a` is not a class scope and `FType` is really, though static, a type *inside* the object `a`. There is no better way to express the link between the instance and the type since we don't know, while writing the code, what the exact type of `a` is, we only know that it matches `Animal`.

Chapter 4

Transformation rules

First, a whole bunch of meta-programming tools that never change, whatever the code, can be put in a separate `.hh` file.

The choice of augmenting a subset of C++ instead of using a wholly different language as input to the transformation process was motivated by the wish to keep the transformation simple. We want the rules to be as local as possible, to allow as much of the code as possible to be passed along. This way, we hope that the rules will remain simple.

4.1 Information gathering

We need to track every class definition as we don't know at which time we will need information about their structure, as inheritance can happen almost anywhere. In other words, the reason for information gathering about class definition is that any class is potentially a base class.

Conversely, when we encounter a child class definition, we don't know if a given method is virtual or not. We have to look into the parent classes' definitions. If the method is virtual and is the first one in the hierarchy (traversing the hierarchy from the top down), it requires then to generate a dispatching method and an implementation. If the method has already been defined upper in the hierarchy, we need only generate the implementation, as the dispatch is handled some parent class.

Similarly, virtual type usage in any place needs the information about virtual type definitions inside classes.

Therefore, we need two things. First, we need some internal data structures to store information about class definitions. Second, we need a way to support namespace separation while storing the information for later retrieval.

4.2 Scope resolution

When specifying base clause in class definitions, we may have to resolve the effective base class referred to. The same need appears when transforming virtual type instantiation statements.

Hence, we need to implement a scope resolution algorithm functionally identical to the one implemented in C++ compilers but handling our internal data structures.

4.3 Implementation in Stratego

The choice of the Stratego language as a program transformation tool arises from the fact that we already have a powerful parser for simple C++ code that can easily be adapted to our "augmented" C++ syntax. The advantage of having a parser is that we can write transformation rules that will be applied on the abstract syntax tree, which is pretty convenient, as its very structure bears part of the needed information. Besides, the standard library is rich enough to allow us to quickly implement a simple working prototype.

The drawback is that the transformation tool will be tightly tied to the C++-flavored languages, as the transformation rules act on the abstract syntax tree that is very C++-specific.

4.4 Transformation examples

What follows is a small set of examples of transformation written in concrete syntax. Because of the complex structure of the standard C++ grammar, a comprehensive set of transformation rules in concrete syntax is beyond our reach right now. Besides, rules alone would not suffice to complete the transformation task. The framework is therefore composed of rules and strategies that combine with each other in order to achieve it.

For a simple base class definition, we would have the following transformation:

```
class A {
  ...
};
```

⟹

```
template <class> class A_;
template <class Inf>
struct vt_trait< A_<Inf> > {
  typedef Inf   InferiorType;
  ...
};
template <class Inf = Bottom>
class A_ : public Void< A_<Inf> > {
public:
  typedef A_<Inf>       selftype;
private:
  ...
};
typedef A_<> A;
```

For a child class definition, the transformation is just a bit more complicated:

```
class B : public A {
  ...
};
```

⟹

```
template <class> class B_;
template <class Inf>
struct vt_trait< B_<Inf> > :
    public vt_trait< A_< B_<Inf> > > {
  typedef Inf   InferiorType;
  ...
};
template <class Inf = Bottom>
class B_ : public A_< B_<Inf> > {
public:
  typedef B_<Inf>       selftype;
  typedef A_<selftype>  supertype;
private:
  ...
};
typedef B_<> B;
```

Here comes the transformation of a function definition which comes in two flavors. The choice is triggered by the introduction of a **`static`** keyword to distinguish between polymorphism over the class hierarchy starting at that class or no polymorphism at all. This construct is somewhat similar to the `class` qualifier in Ada. It is an accessory feature that is so easy to implement that it would be a shame not to use it.

```
sometype foo(A& a)
{
  ...
}
sometype bar(static A& a)
{
  ...
}
```

$\Longrightarrow$

```
template <class T>
sometype foo(A<T>& a)
{
  ...
}
sometype foo(A& a)
{
  ...
}
```

Chapter 5

Conclusion

5.1 Project status and perspectives

A first prototype is currently under development. Its purpose is to set the general transformation framework that is to be extended in future versions.

The planned features are the transformation of class hierarchies to static recurring flavor with support of **`selftype`** and **`supertype`** and the **`static`** qualifier for function arguments.

Then, pretty quickly, we want to implement support for virtual types and parametric polymorphism. The problem with parametric polymorphism is that the C++ parser we intend to use is not ready for parsing templates. Its author's goal was to write a C++ parser able to validate C++ code with respect to the standard grammar, but this requires unfortunately the implementation of the template instantiation mechanisms which is a pretty tedious task. We plan to bend the C++ standard grammar to circumvent this requirement and parse templates in a slightly modified way.

On a longer schedule, we plan to support F-bounded and match-bounded parametric polymorphism and some additional sugar that is not so urgent for the moment (conditional inheritance, polymorphism on template parameters, etc).

The ultimate plan is to be able to rewrite generic algorithms in the Olena project.

5.2 Personal conclusion

Much reading was necessary to grasp the essence of what we wanted from theory to be supported. Besides, the literature often sinks into deep formalism which is not necessarily easy to link to concrete imperative language constructs.

The field was new for me and allowed me to embrace a lot of the current challenges in modern OO languages and applications.

Next steps seem very interesting as they comprise diving into development in the Stratego language and tighter collaboration with other members of the LRDE.

Chapter 6

Bibliography

[1] *International Standard of C++*, September 1998. ISO/IEC 14882:1998(E).

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science Series. Springer-Verlag, 1996.

[3] Robert Anisko. Program transformation and the C++ language. Technical report, LRDE, October 2002. Séminaire LRDE.

[4] Kim B. Bruce. Typing in object-oriented languages: Achieving expressibility and safety. Tutorial T12 at the 12th European Conference on Object-Oriented Programming (ECOOP), July 1998.

[5] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the International Conference on Functional Programming and Computer Architecture*, pages 273–280, London, UK, September 1989. ACM.

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[7] Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

[8] Raphaël Poss. Techniques for implementing oo paradigms in static c++. Technical report, LRDE, May 2002. Séminaire LRDE.

[9] Nicolas Tisserand. Tools for C++ programs transformations. Technical report, LRDE, September 2002. Séminaire LRDE.

[10] Mads Torgersen. Virtual types are statically safe. In *Proceedings of the 5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, San Diego, CA, January 1998.

[11] Erwin Unruh. Prime number computation, 1994. ANSI X3J16-94-0075/ISO WG21-462.

[12] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[13] Todd L. Veldhuizen. Techniques for scientific C++, August 1999.

[14] E. Visser. The stratego library, 2000.

[15] E. Visser. The stratego reference manual, 2002.

[16] E. Visser. The stratego tutorial, 2002.