A static and complete object-oriented model in C++
mixing benefits of traditional OOP and static programming

Nicolas Burrus, Thierry Géraud, David Lesage, Raphaël Poss

EPITA Research and Development Laboratory (LRDE)
14-16, rue Voltaire - F-94276 Le Kremlin Bicêtre cedex - France
{nicolas.burrus,thierry.geraud,david.lesage,raphael.poss}@lrde.epita.fr

This work has its origin in the development of Olena, a C++ generic image processing library from the LRDE [2, 11]. As in any scientific, object-oriented and real-scale software, we faced both modeling and performance issues that led us to study C++ features with deeper insight. We finally expressed the need for mixing object-oriented programming and static genericity in C++. Since many static programming techniques already exist, we attempted to formalize and to enhance their usage. In this paper, we present a model that provides all the features of the object-oriented paradigm – and some additional ones – and benefits from the performance of static programming.

1 C++: a multiparadigm language

The C++ language provides two different typing systems:

– a usual object-oriented typing system, based on subclassing by inheritance;
– a structural-like, unbounded typing system based on the *template* construct.

These two frameworks have different purposes and thus different benefits and drawbacks. They eventually lead to different programming paradigms: classical OOP paradigm for the former, generic and static paradigms for the latter. So far, if these paradigms have been deeply studied separately, combining them an optimal way remains problematic.

1.1 On the object-oriented paradigm in C++

The usual C++ object-oriented paradigm allows developers to design advanced software through a classical inheritance system and convenient features like coercion, overloading and inclusion polymorphism.

Unfortunately, these well-known mechanisms have severe drawbacks. Overloading and inclusion polymorphisms imply costly dynamic bindings which are unacceptable in heavily computational applications. See Figure 1 for a typical example where such a virtual function binding occurs.

Different approaches have already been studied in order to face this runtime cost common to most object-oriented languages [19, 10]. More generally, static analysis [4, 3] and partial evaluation [13] can be applied to object-oriented languages to optimize virtual function calls. Unfortunately, almost all these efforts remain research projects.

1.2 On the static and generic paradigms in C++

The generative power of C++, based on the *template* construct, permits parametric polymorphism and then higher code factorization and reusability. Mixed with overloading and specialization mechanisms, parameterization enables developers to express both generic and specialized implementations. This comes with limited efficiency loss since parameter matching in C++ is performed statically. Some widely adopted C++ libraries have proved the

```
// 'image' is an abstract class
void foo(const image& ima)
{
    // ...
    ima.m(); // 'm' member function is virtual in 'image'
    // ...
}
```

Fig. 1. Classical OO implementation of polymorphic algorithms.
The call to the method `m` requires a costly dynamic dispatch to find the actual implementation. However, the signature of `foo` is strong, nothing but a compatible type of `image` will be accepted.

workability of such a polymorphism, like the Standard Template Library (STL, see [16]) and Boost [1].

Yet, classical, STL-like parameterization in C++ remains an unbounded typing system, despite underlying, implicit "concepts" [16]. There is no explicit requirement on the parameterized types, so that we cannot define two generic functions with the same name and the same arity. Therefore, such a weak, structural-like typing system cannot be entirely satisfactory. Several works already tried to cope with this lack of explicit constraint on the parameters in C++, introducing for example static concept checking [12, 14] through language extensions.

Another limitation is the exact matching of parameters that makes mixing inclusion polymorphism and specialization impossible. Then, subclasses of a class `A` cannot benefit from a specialization of an algorithm on `A`.

STL-like programming drawbacks are illustrated in Figure 2.

```
// Parameter 'Image' is a free type variable.
template<class Image>
void foo(const Image& ima)
{
    // ...
    ima.m();
    // ...
}

// The following version, specialized on 'image2d' image type
// is not eligible for subclasses of 'image2d'.
template<>
void foo(const image2d& ima)
{
  // ...
  ima.m();
  // 'image2d'-specialized treatment
}
```

Fig. 2. STL-like implementation of polymorphic algorithms.
No dynamic dispatch is required to call the `m` method. However, the signature of `foo` is weak and overloading rules will choose the second version of `foo` only for exact image2d parameters.

1.3 Our objectives

Object-oriented mechanisms, inheritance, inclusion polymorphism, overloading and labeled types are highly valuable for software designers. More generally, we do not want to lose the object orientation of classical C++ programming. We also seek the performance and the specialization capabilities of the parametric polymorphism, but with stronger typing.

To achieve such an aim, different approaches can be considered. First, we could extend the C++ language with mechanisms like static, bounded parameterization, as it was proposed for Java in [9] [1]. The other alternative is to stay within the bounds of the C++ language without any extension, since it natively provides object-oriented mechanisms and static parameterization. As library developers, we had to rule out the former one because we do not want our final users to need any additional equipment.

In the following sections, we show how we managed mixing static, generic programming and the classical OOP thanks to different existing techniques. We eventually show that our work succeeded in defining a full-static and complete object model, with additional properties w.r.t. type-control and design capabilities.

2 Description of the model

To achieve a full-static model, we had to simulate or to assist traditional OOP mechanisms with static programming techniques. The main point was to design a static inclusion polymorphism system and then to adapt the inheritance idiom.

2.1 Static hierarchies

The core of our model is a static hierarchy derived from the Barton & Nackman trick [5]. In [18], Veldhuizen had already discussed some extensions of this technique and assumed the possibility of applying it to hierarchies with several levels. We effectively managed to generalize these techniques to entire, effective hierarchies.

In our model, non final classes [2] are parameterized by the exact type of their most derived class. Additionally, any class hierarchy must inherit from a special base class called `any`. This class provides an `exact()` accessor to downcast the object instance to its actual concrete type. Figure 3 describes the kind of hierarchy we obtain.

Effectively, the class parameterization implies that for any hierarchy involving N leaf classes, N distinct types for base classes are instantiated. Therefore, dynamic polymorphism over the base classes is unfortunately impossible.

2.2 Abstract classes and interfaces

The abstraction power comes from the ability to express class interfaces without implementation, as in Java. Our model keeps the idea that C++ interfaces are represented by abstract classes. Instead of defining purely virtual member functions, abstract classes define abstract member functions as dispatches to their actual implementation. This manual dispatch is performed thanks to the `exact()` accessor provided by the `any` class. An example of abstract member function is given in Figure 4.

The compliance to a particular interface is then naturally ensured by inheritance from the corresponding abstract class.

[1] in [9], Day introduces static and bounded parameterization for Java. So-called *where clauses* constructs allow both structural and label constraints on parameters.

[2] Non final classes are abstract classes or concrete classes that can be extended. Non parameterized classes are necessarily final.

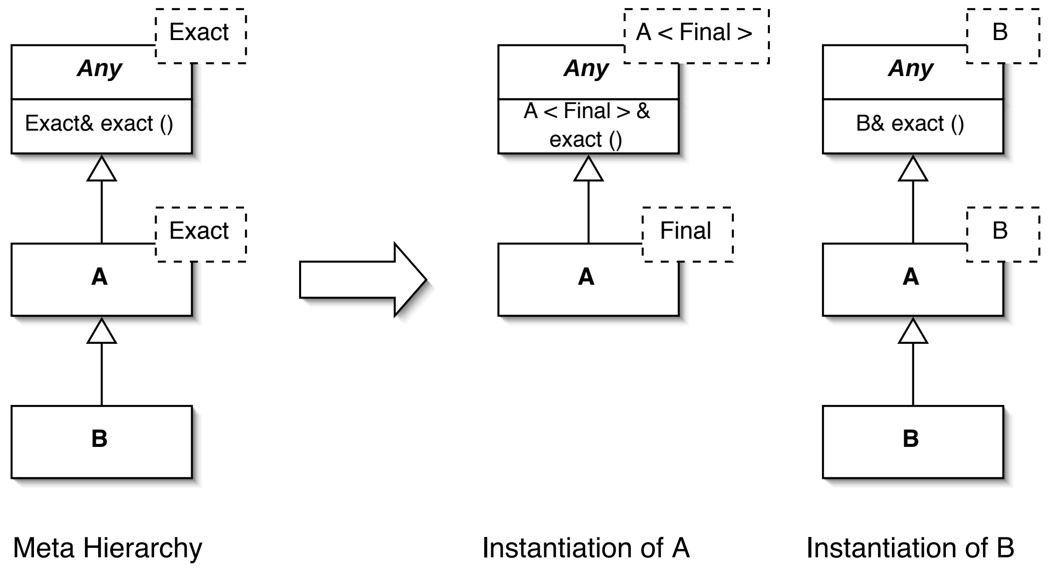

Fig. 3. UML description of the model.
A single meta hierarchy generates one class hierarchy per instantiated class. Our model can instantiate both leaf classes and intermediate ones.

```
template <class Exact>
RetType foo<Exact>::m(args ...)
{
  return this->exact().m_impl(args ...);
}
```

Fig. 4. Abstract member function sample

3 Formalizing the model properties

Since it mixes OOP and structural programming, our paradigm inherits two main categories of properties. This results in a full-featured object model and, in the meantime, in a strong-typed, high performance, flexible and statically checked model.

3.1 A complete object model

We preserved the full expression power of the standard object-oriented paradigm:

- complex object hierarchies, even with multiple inheritance and diamond constructions, can be deduced directly from the modeling;
- interfaces and abstract classes can also be implemented by explicitly preventing their instantiation;
- we rely on the C++ inheritance mechanisms, so procedural algorithms can use both specialization and overloading facilities without modifications;
- methods can be selectively defined in the subclasses, default implementations in base classes will be called otherwise;
- our manual dispatch has enough information (the exact type) to handle method overriding in the same way than traditional C++ does.

3.2 A strong-typed and efficient model

In our model, methods and functions can express typing requirements over class hierarchies using an hybrid between label typing and static genericity. In fact, our model introduces a new bounded and labeled typing system. Figure 5 gives a use-case of a procedure implementation within this typing scheme. The concrete type `I` remains a free type variable, but it is constrained to be a subclass of `image<I>`. This parametric polymorphism offers much more type control than the STL programming style (see Figure 2), thanks to explicitly labeled template types.

Moreover, this control gain comes with no performance loss compared to STL-like programming. Typing remains entirely static, so that static dispatch is still possible.

```
// 'image' is an abstract template class.
// Parameter 'I' is constrained to be a subclass of 'image<I>'
// so that 'I' implements 'image' generic interface.
template <class I>
void foo(const image<I>& ima)
{
  // ...
  ima.m(); // call to 'm' implementation is dispatched statically
  // ...
}
```

Fig. 5. Our implementation of polymorphic algorithms.
The signature of `foo` is labeled and strong-typed. Dynamic dispatch is avoided.

3.3 A flexible but statically checked model

Subclassing in our paradigm is much more flexible than traditional subtyping. We are actually quite close to F-bounded polymorphism [8] and matching bounded polymorphism [6]. Indeed, when we write `template <class I> void foo(image<I>&)` , we require that `I` conforms to `image`, which is a type function, applied to `I`. *Type conformance is ensured by C++ inheritance at compile-time.*

In addition, parent classes have access not only to the exact type, but also to properties related to the exact type thanks to traits [18]. This feature, combined with the flexibility of the typing and with compile-time type checking makes several interesting constructions possible in a safer way. We now detail three relevant examples: statically checked virtual types, method argument covariance with static checks and multi-methods with static dispatch.

Statically checked virtual types Statically checked virtual types are really useful in object-oriented design, as stated in [7]. Within our paradigm, it is possible to use bounded virtual types that are checked at compile-time. Figure 6 illustrates this mechanism. Note that not only types but also constants can be defined virtually using the same mechanisms.

```
template <class P>
struct point : public any<P> {};

struct point2d : public point<point2d > {};

template <class I>
struct image : public any<I>
{
  void set(Point<get_exact_point_type(I)>& p)
  {
    exact().set(p.exact());
  }
};

define_exact_point_type(image2d, point2d);
struct image2d : public image<image2d>
{
  void set(point2d& p) { ... }
};
```

Fig. 6. Statically checked virtual type
`get_exact_point_type` and `define_exact_point_type` are simple macros hiding traits definitions. In this example, nothing but a `point2d` can be given to `image<image2d>::set` without a compile-time failure. It is important to notice that the virtual point type must conform to the `point` type function.

Statically checked argument covariance Covariant parameters are usually wanted when modeling with objects. This cannot be done in basic C++ using virtual methods. Type safe covariance was already studied using templates in [17]. Our approach is quite different and simpler since we have a stronger and more flexible type system. Using our paradigm, it becomes possible to easily write methods with covariant parameters, with failures at compile-time if misused. This is demonstrated in Figure 7.

```
template <class I> struct image : public any<I>
{
  template <class P>
  void set(point<P>& p)
  {
    exact().set(p.exact());
  }
};

struct image2d : public image<image2d>
{
  void set(point2d& p) { ... }
};
```

Fig. 7. Statically checked argument covariance
If a `point3d` is given to `image<image2d>::set`, failure will occur at compile-time since the compiler will not find any method `image2d::set` accepting `point3d` arguments.

Multi-methods with static dispatch In most of the popular object-oriented languages, only one dispatch can be achieved on the actual type of an argument. Several methods have been studied to improve the C++ with multi-method dispatch, as in [15]. With our paradigm, since we can get the exact type of an argument statically, we can do manual multi-method dispatch easily, as shown in Figure 8.

```
template <class I1, class I2>
void foo(image2d<I>& i1, image3d<I2>& i2);

// other versions of foo ...

template <class I1, class I2>
void bar(image<I1>& i1, image<I2>& i2)
{
  foo (i1.exact(), i2.exact()); // dispatch is helped here
}
```

Fig. 8. Static dispatch for multi-methods
`bar`, by giving exact instances of arguments, permits the compiler to find the good function considering the exact types of i1 and i2, thus emulating a multi-method dispatch.

4 Conclusion

In this paper, we described a new complete and full-static object-oriented model in C++. This model combines the expression power of traditional OO and the performance of static programming. It introduces a strong-typed but flexible typing scheme that brings new interesting features such as statically checked virtual types and argument covariance.

This paradigm has been implemented and successfully deployed in Olena. The library mixes different complex hierarchies (images, points, neighborhoods) and heavily relies on overloading capabilities. We make an intensive use of virtual types and multi-methods, which considerably simplify the expression of generic algorithms and increase the overall type safety.

The main limitation of our model is the closed world assumption, which may be inadequate for some projects since it prevents the usage of separated compilation and dynamic libraries. The other issues are common drawbacks of the intensive use of templates:

– heavy compilation time;
– cryptic error messages;
– unusual code, unreadable by the casual reader.

However, the model and the collection of associated constructs are suitable (at least partially) for most projects. Therefore, it can be perceived as an alternative to the traditional object-oriented paradigm when both performance and design capabilities are critical.

References

1. Boost libraries. http://www.boost.org.
2. Olena image processing library. http://www.lrde.epita.fr/olena.
3. Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. *Lecture Notes in Computer Science*, 1098:142–167, 1996.
4. David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, pages 324–341, 1996.
5. John Barton and Lee Nackman. *Scientific and engineering C++*. Addison-Wesley, 1994.
6. Kim B. Bruce. Typing in object-oriented languages: Achieving expressibility and safety, 1996.
7. Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549, Brussels, Belgium, July 1998. Springer-Verlag.
8. Peter S. Canning, William R. Cook, Walter L. Hill, John C. Mitchell, and Walter G. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 73–280, London, UK, September 1989. ACM.
9. Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Meyers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, SIGPLAN Notices volume 30 number 10, pages 156–168, 1995.
10. David Detlefs and Ole Agesen. Inlining of virtual methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*, Lisbon, Portugal, June 1999.
11. Alexandre Duret-Lutz. Olena: a component-based platform for image processing, mixing generic, generative and OO programming. In *Proceedings of 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE 2000)*, pages 653–659, Erfurt, Germany, October 2000. Young Researchers Workshop (published in "Net.ObjectDays2000").
12. Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.
13. Ulrik P. Schultz. Partial evaluation for class-based object-oriented languages. *Lecture Notes in Computer Science*, 2053:173–198, 2001.
14. Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, October 2000.
15. Julian Smith. C++ & multimethods. *ACCU spring 2003 conference*, 2003.
16. A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, 1994.
17. Vitaly Surazhsky and Joseph Y. Gil. Type-safe covariance in C++, 2002. Unpublished.
18. Todd L. Veldhuizen. Techniques for scientific C++, August 1999.

19. Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *12th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*, volume 32 of *Issue 10*, pages 125–141, Athlanta, GA, USA, October 1997.