Abstract: Software performance benefits from executing computations on constant data statically, at compile time. Unlike most languages, C++ allows expressing static computations by language means (template meta-programming), without relying on separate pre-processing tools.

However, static code differs from dynamic code by syntax and even by programming style, as the former is functional, the latter imperative. A library designer who wants to support both a static and a dynamic version of an algorithm is therefore forced to design and maintain two implementations.

A common formulation is presented that allows one to implement an algorithm generically by a single definition, while encoding static parameter properties within the argument types. By the choice of argument representation, the user determines how the algorithm is evaluated. This decoupling preserves all information necessary to have the compiler create appropriate code for each case.

The realization by an ANSI C++ library of static constructs guarantees portability and allows an optimizing compiler to eliminate run-time overhead.

Keywords: generic programming, meta programming, staging, code optimization, library-based software, C++

1 Motivation

Modern software is characterized by fine-grained, highly reusable components that are easily, flexibly, and efficiently combined to specify particular algorithms or data structures, which again form components themselves.

Unfortunately, things are not quite like that yet! Granted, generic programming methods introduce a high degree of abstraction by decoupling algorithms from data structures using uniform function invocation syntax. But they still rely on separate, explicit function implementations to encode the evaluation of an algorithm according to state-related properties of its parameters.

For instance, an algorithm may be evaluated statically, if given static arguments, or dynamically otherwise. Most programming languages only allow expressing the latter or require a special syntax to express the former.

What is a library designer supposed to do, thus? Providing syntactically different implementations to support both static and dynamic computations, is tedious and error-prone. In practice, almost all library designers therefore end up supporting just one case—the more common, but also slower dynamic version. Such limited support contradicts the philosophy of performance-oriented generic programming a la C++ Standard Template Library (STL) [13] and its derivatives, which advocate the decomposition of code into boiled-down algorithms and data structures.

To avoid the drawbacks of code replication, an alternative, uniform formulation of generic algorithm implementations will be presented. After a closer look at different examples, requirements on a solution are stated in Sect. 2. The essential ideas and elements of the chosen solution are explained in Sect. 3 and illustrated in Sect. 4. Finally, Sect. 5 summarizes and discusses the results.

Although the presented solution does not claim to form a fundamentally new approach, it demonstrates how to systematically combine existing methods to achieve the desired genericity at the implementation level.
1.1 Inconsistent Syntax

In a stateful language like C++, the Euclidean algorithm for computing the greatest common divisor (gcd) of two integral values (generally: elements of a Euclidean domain) at run time is usually implemented as in Ex. # 1 and invoked as in Ex. # 4.

If the arguments are constant, though, the computation can be performed at compile time, given its alternative implementation by a template metafunction that realizes the same algorithm in a different style, but that employs template specialization and recursion rather than a loop, as shown in Ex. # 2.

The invocation of this version, given in Ex. # 5, has a syntax different from its dynamic counterpart. Yet another implementation is required if one wants to support partial evaluation in the case that just one argument is constant. Again, the invocation of that version comes with its own syntax.

This multiplicity of style, implementation and call syntax leads implementors to support merely one kind of implementation.

1.2 Temporary Data

Another issue in the discussion of implementation alternatives is the question whether, in the dynamic case, arguments are passed by reference or by value, or, equivalently, whether the values of the actual parameters of a function call may be modified or not. In C++, both ways are feasible, but require separate implementations, which differ only by their formal parameter types, see Ex. # 3. Here, incidentally, the modifying invocation, shown in Ex. # 6, is syntactically identical to Ex. # 4, given that its arguments are lvalues.

Argument modification might seem a minor issue for basic argument types. It allows crucial optimizations, though, for more complex, user-defined types, e.g., polynomials.

1.3 Late Binding

While the C++ language itself does not provide means for expressing late binding known from most functional languages, the STL, whose algorithms strongly rely on strategies expressed by functions or functors, raised interest in having constructs to define functors ad-hoc. Libraries like FC++ [8, 12] or the Boost Lambda Library (BLL) [5, 6] provide constructs to form partially bound expressions, which the Boost Metaprogramming Library (MPL) [1, 4] adopted for static expressions.

But, these libraries impose additional requirements on function definitions, which need to provide, e.g., return type information. As before, further implementation complexity makes it harder for the application developer to support different possibilities of function evaluation.

In conclusion, the three example classes just described should illustrate two points: First, there are various ways to implement a given algorithm. Second, the major obstacle to providing a uniform implementation syntax prove to be explicit assumptions of parameter properties within those implementations.

2 Goals

To overcome the multiplicity of formulations, a generic syntax for function implementations and invocations is needed that encapsulates realization details like those exemplified in Sect. 1.

For each function invocation, the generic implementation has to be translated to a specific realization that accords with the actual arguments’ properties. This translation should be performed at compile time to avoid run-time overhead.

A necessary prerequisite is the encoding of argument properties within the argument types, while the formal parameters are free, to make the implementation independent of those properties.

2.1 Generic Implementations

As motivated, the implementation of an algorithm shall be uniform, canceling out all explicit references to particular argument properties. The algorithm has to be specified in a format that is applicable across different programming styles and parameter semantics. The underlying details, that actually cause appropriate code to be produced when compiling the definition for different parameter properties, are encapsulated within elementary functions.
conventional approach ⇒ multiple specific formulations

```c
# 1
inline int gcd(int a, int b)
{
 while (b!=0)
  {
    int r = a % b;
    a = b;
    b = r;
  }
 return a;
}
```

```c
# 4
int main()
{
 int x(18), y(30);
 return gcd(x,y);
}
```

dynamically evaluated, non-modifying

```c
# 2
template <int A, int B>
struct Gcd : Gcd<B,A%B>
{};
template <int A>
struct Gcd<A,0>
{
    enum { RESULT = A };
};
```

```c
# 5
Gcd<18,30>::RESULT
```

[statically evaluated]

```c
# 3
inline int& gcd(int& a, int& b)
{
    // ...
}
```

```c
# 6
gcd(x,y)
```

dynamically evaluated, modifying

<table>
<thead>
<tr>
<th>implementations</th>
<th>invocations</th>
</tr>
</thead>
<tbody>
<tr>
<td># 7</td>
<td># 8</td>
</tr>
</tbody>
</table>
gcd := λ x y. ((while (λ t. not (isNeutral t2)) (λ t. tuple t2 (mod t1 t2))) (tuple x y))1 |
gcd(save(x),save(y))

dynamically evaluated, non-modifying

```c
# 9
gcd(i(18),i(30))
```

[statically evaluated]

```c
# 10
gcd(x,p(1))(y)
```

[partial binding, dynamically evaluated]

[suggested approach ⇒ single generic formulation]
The implementation may not rely on the arguments being static or dynamic, nor on them being passed by reference or by value, nor should states be expected to exist at all. E.g., iterations have to be expressed independently of their realization by recursion or loops. As a consequence, the formulation has to be semantically independent of the underlying implementation and conceptually even of the language, hence reduced to a rather mathematical formulation that merely expresses the algorithmic steps.

This idea is illustrated by the (non-C++) formulation of gcd in Ex. # 7, which expresses the Euclidean algorithm in such abstract way. Note that the example does not mean to introduce a notation, but to demonstrate how the definition would have to be formed in terms of more elementary functions which encapsulate implementation details. In the example, few functions are needed, particularly:

`while` has the behavioral purpose of abstracting an iteration. It takes a termination condition and a function to invoke in each iteration. As the different implementations in Exs. #1 & 2 motivate, `while` is specialized to a recursion in case of a static argument, and to an imperative loop if the argument has dynamic state. If in the latter case the argument is not modifiable, a modifiable copy has to be created, to which updates are applied, and which is then returned by value.

`tupla` has the technical purpose of creating an aggregate that contains its arguments. Conversely, a subscript index symbolizes retrieval of the according element from the indexed aggregate.

`isNeutral` has the logical purpose of telling whether its argument is the neutral element (of the according Euclidean domain).

### 2.2 Generic Invocations

Since any explicit reference to parameter properties has been removed from the generic implementation, this information has to be encoded completely within the arguments—more precisely, within their types, to be available at compile time. The user can freely choose the argument properties when invoking a function, while the automatic selection (or creation) of the appropriate realization of its implementation is performed statically.

Therefore, means for expressing object properties are required, which, in the scope of this paper, extend to constant or modifiable and static or dynamic state, and to instant or late binding. For each of the three pairs, it is convenient to identify some natural default behavior and to require users to explicitly specify only its alternatives.

Starting with modifiability, C++ already provides an encoding of this property by the qualifiers `const` and `volatile`, which form part of type declarators. The default mode of passing an argument to the generic `gcd`, as in Ex. # 4, should preserve its modifiability. This decision is not only natural, but also necessary, as this property can only be changed towards stricter qualification, as in the invocation for decidedly non-modifying evaluation in Ex. # 8. The encapsulation by `save` spares the user the corresponding type casts to protect the arguments.

For the option of static or dynamic evaluation, second, it suffices to supply this information explicitly for such arguments that have static state, as suggested in Ex. # 9. Note that `i` must be a preprocessor macro rather than a function, which would consider its argument a dynamic `int` value. It (lexically) creates an argument whose type represents the given static integer value by common techniques [2]. Invocations by both a static and a dynamic argument may then automatically cause partial evaluation—the Euclidean algorithm, though, is a well-known example for this not to happen [3].

Finally, for the choice between instant and late binding, a function should be partially bound if an argument is supplied that explicitly indicates to have late binding and therefore to act as a free variable. In Ex. # 10, the argument `p(1)` would be replaced by the first argument `y` when the partially bound result of the invocation is completely bound.

### 3 Approach

Crucial for the workings of a generic implementation is a mechanism for two-stage function binding and evaluation. First, when invoking a two-stage function, its implementation is evaluated statically. Then, the result should be an appropriate realization of the function to be evaluated dynamically. In the following, we elaborate a method to provide this mechanism.
3.1 Unifying Implementations

The existence of an embedding of an argument’s static properties, like modifiability, binding, or possibly its value, in dynamic context is essential if one wants to be able to use arguments of any property commonly in expressions. The common format must be dynamic, as this is the “weaker” stage: Static data can be mapped bijectively to a set of dynamic objects by performing a mapping to a set of types (e.g., by instantiating a designated template) and creating instances of these [2], but not vice versa, naturally. An inversion of the embedding is necessary to retrieve the property information from arguments in function invocations.

Dynamic arguments created by the embedding are usually empty and of trivial semantics, which allows the compiler to eliminate run-time overhead of this abstraction. In illustration, Exs. #8–10 display the three different embeddings save, i, and p, which encode the properties of immutability, static evaluatability, and late binding, respectively. As the example invocations show, the corresponding results of the embeddings serve as arguments in exactly the same way as the usual run-time arguments. The results of i(18) and p(1) are objects of empty classes and may therefore be eliminated.

The implementation of an algorithm has to be provided generically, to accept arguments with different properties and to allow its uniform invocation by generic function calls. This is the essential part of the solution, which relies on a two-stage representation of function implementations.

A function invocation delegates its arguments to the two-stage representation of its implementation. First, it binds and evaluates the implementation statically with the argument types. Since properties are encoded as types, they can be detected at compile time and the appropriate run-time code be produced.

Then, the resulting dynamic function definition is bound with the arguments of the invocation at run time and passed to the C++ run-time system. If all arguments have static state, the static evaluation has already determined the (static) result, and the evaluation of the corresponding dynamic realization of the implementation just returns the dynamic embedding of the result. Static state is preserved, which allows the composition of static computations.

3.2 Generating Realizations

Given the principle of two-stage evaluation of function invocations, a systematics of implementing generic two-stage functions is needed so that the different specialized realizations can be created automatically. Otherwise, users would still have to provide each time a new and complete set of all appropriate realizations of an algorithm for any combination of argument properties. In the interest of automation, we therefore expect a compositional style of function definition in the spirit of Ex. #7.

Given a generic function implementation and its invocation with arguments of statically given properties, the compiler can then create the appropriate realization on the fly, by composing the according realizations of the component functions. This recursion ends where the components are no compositions themselves, but atoms. If just for these elementary functions appropriate realizations exist, the correct realization of the invoked function is created inductively and automatically.

For a base, realizations for those atomic functions have to be provided, which are built-in operations and other non-decomposable functions. This is a large, but limited task.

The implementation of functions by functional composition requires a representation of unbound parameters, i.e., free variables. These form the simplest case of partially bound functions, whose representation is commonly expressed in C++ by wrapping the function and the arguments in an object.

In conformity with the principle of encoding properties within arguments, the presence of a partially bound argument in a function invocation implies that its evaluation is delayed, i.e., that a new partially bound function is created.

This technique has been similarly applied in FC++, BLL and MPL and is therefore not outlined further. In contrast to these applications, the delaying of binding and evaluation is generalized here to refer to both the static and the dynamic stage.

Free variables provided, the explicitly realized elementary functions can be composed to more complex functions in a very natural and convenient way. Thanks to the simultaneous consideration of both the static and the dynamic state, the composition automatically provides proper static and dynamic definitions. There is no need to provide realizations of complex functions from scratch!
3.3 Technical Issues

A few technical remarks remain to be stated. First we note that for function terms (in contrast to object terms) the static formulation is weaker than the dynamic one, as any dynamic function has a static aspect while the opposite not necessarily holds. Therefore, the composition has to be expressed statically, which is accomplished in C++ by instantiating class templates.

Since the C++ template meta-programming language has pure value semantics, function definitions have to follow a functional programming style under the restrictions of C++ syntax. Otherwise, we would have to devise a separate preprocessor.

All constructs have to be designed in a way that avoids run-time overhead in comparison to a direct implementation. For that, we use common techniques, including the common rule of statically expressing everything that has static context, though refraining from all constructs that, according to the language definition, necessarily cause run-time overhead.

While the success of this approach depends on the compiler’s optimizing capabilities of eliminating unnecessary abstraction overhead, particularly of the dynamic embeddings of static data, it does not introduce any principal obstacles to entire elimination of additional costs.

4 Realization

To illustrate the approach described in Sect. 3, we present our realization in C++ by detailing the different steps that are performed when a generic algorithm is invoked. For a running example, we return to Ex. #8, where the generic gcd is invoked with dynamic and non-modifiable arguments. The code examples are simplified for reasons of comprehensibility, rather than to faithfully reflect the actual realization [9], which has to deal with many merely technical matters of C++.

Before gcd can be invoked, it must be made known to our library. We therefore have to first “register” the function identifier and to describe the corresponding interface, along with some characteristics of the body. Technically, registration happens through a macro that causes the preprocessor to generate code that lexically depends on the function identifier introduced. In the example, the macro call

\[
\text{DECLARE}(\text{Gcd}, \text{gcd}, 2, \text{SPECIAL}, \text{DYNAMIC}, \text{INSTANT})
\]

has to be supplied within the file DECLARE.hh. It provides identifiers for static and dynamic code elements, and a description of the expression semantics of gcd, which:

- takes 2 arguments,
- may be \text{SPECIAL}ialized for user-defined types,
- is evaluated \text{DYNAMIC}ally, and
- allows \text{INSTANT} binding.

As the library is given by static ANSI C++ code, just a header file has to be included in applications to use it.

```
#include "cong.hh"
using namespace cong;
```

The invocation given in Ex. #8 initially invokes save to create constant references to x and y, and passes these as arguments to the (overloaded) generic function gcd which is created by a DECLARE macro call.

```
template <typename S1_, typename S2_>
inline
typename Gcd<Exp<S1_ const&>,
    Exp<S2_ const&> >::Result
gcd(S1_ const& param1, S2_ const& param2)
{
    return Gcd<Exp<S1_ const&>,
                Exp<S2_ const&> >::result(param1,param2);
}
```

The instantiation of gcd enforces the instantiation of the class template Gcd, of which it retrieves both its return type and the function to which to delegate the call.

```
template <class Exp1_, class Exp2_>
struct Gcd
: Bind
    <Exp<Internal<STATIC,BOUNDED,, >, 
    Named<SPECIAL,2ul,GcdTag> > >,
    Exp1_,Exp2_>
{};
```
Gcd itself inherits this information from an instance of the template class Bind, providing a static first-class representation of gcd as its 0th argument.

Bind statically analyzes the argument properties by template meta-functions and, if applicable, selects a generic realization, e.g., to delay evaluation in presence of an unbound argument.

In the case of Ex. # 8, however, it delegates evaluation to the implementation of its 0th argument, representing gcd. This implementation is generically defined by functional composition as described in Sect. 3.

```
template <>
struct Define<GcdTag>
: As<Get<Bind<Whilst<BoolNot<IsNeutral
  <Get<P<1>,I<2> >
  > >,
  Tuple<Get<P<1>,I<2> >,
  Mod
  <Get<P<1>,I<1> >, Get<P<1>,I<2> >
  > > >,
  Tuple<P<1>,P<2> > >, I<1> >
  }
{
}
```

Note how closely the generic implementation resembles the pseudo code in Ex. # 7, if one disregards C++ meta-language syntax.

Evaluating the implementation of gcd requires evaluating all components of the implementation, glued by partial binding. At some point, Whilst is bound with two partially bound two-stage functions. The result of this binding is a unary two-stage function, and, finally, atomic, which means that realizations of this function are explicitly defined to specify how to perform the iteration.

As the function is bound with a tuple that contains references to \(x\) and \(y\), its realization for a dynamic argument is chosen, which performs a loop. For simplicity, just the core of this (relatively complex) realization is displayed, which forms the outcome of the last-mentioned binding (\(\text{param}_1\) is the tuple).

```
typedef typename Modifiable<Exp<Param1_> > ::Result Result;
static Result result(Param1_ param1)
{
  Result r(param1);
  while (toBoolean(bind(Fun1_(),r)))
  {
      r = bind(Fun2_(),r);
  }
  return r;
}
```

If the argument is not modifiable—which applies in the example, where it is a tuple of non-modifiable elements—the variable \(r\) is defined to be a modifiable representation of the argument. Note that in the example, this representation can not be a mere copy of the tuple, but has to be a tuple of (modifiable) int lvalues. If the argument is modifiable, the variable \(r\) is just chosen to be a reference to the argument.

At this point, the inductive creation of a realization of gcd has reached a leaf of the composition tree, which is traversed in depth-first order. The compiler passes this code to the C++ run-time system, after having created the tuple argument by previous invocations of appropriate realizations of leaves of the subtree of the Bind node. The resulting tuple is passed to the parent node, which extracts the first element of this tuple. Virtually, the compiler has hereby composed a realization of gcd for arguments of the given properties.

We conclude the section with a brief description of the library as a whole. Besides the already mentioned components, the library presently covers essentially all C++ built-in types, conversions, and operations. It also supports fundamental type properties and relations (“traits”); in addition to the ones we have seen, these properties include type identification and classification, derivation of dependent types and type conversions. Finally, a small number of elementary functions is available, mainly the creation of objects, conversions, conditions, and the quoting of late binding.

The core of the library consists of few dozens of template classes, many of which are trivial, but some of which are specialized in a large number of ways to provide function- or property-dependent realizations of atomic implementations. These specializations form the major part of the about 3000–4000 lines of code, where preprocessor facilities are exploited to avoid lexical repetitions [9].

5 Conclusion

Conventional generic C++ programming techniques are hardly applicable to the level of algorithm im-
implementations. This restriction severely compromises the development of flexible, performance-oriented software.

We therefore looked for a way to generically implement algorithms, without reducing the set of actual realizations. The proposed solution is founded on a strict separation of concerns: each actual function argument encapsulates all properties the client code wants to be taken into account when the function body is evaluated. Released from any responsibilities for expressing the semantics of program parameters, conversely, functions can be uniformly represented in a fully generic, two-stage manner. The underlying machinery we provide selects the actual realization of the generic implementation both statically and based upon the parameter properties. The staged approach facilitates the systematic application of partial evaluation methods [7].

By generalizing previous implementations of partially bound functions in FC++, BLL, or MPL to two-stage functions, algorithms can be conveniently implemented as functional compositions of simpler algorithms. These compositions have the crucial effect of forcing the compiler to inductively create appropriate realizations for given argument properties, given that such realizations are provided for the atomic algorithms forming the leaves of the composition tree.

The solution allows one, as intended, to design generic algorithms “more generically”. While implementation efforts are reduced, local optimizations are automatically propagated across the code.

In multi-stage languages [14] like MetaML [10] or Template Haskell [11], staging methods methods are directly supported by compilers, while other languages require separate tools to perform staging. The presented C++ solution, in contrast, has the charm of being provided by a portable and modifiable library. Consequently, this design allows the integration with other libraries.

5.1 Applications

The presented solution opens the door to various optimization methods, from the automatically performed partial evaluation of static subexpressions to possible transformations based on static program analysis. Here, the library approach offers the particular advantage of allowing the user to supply auxiliary information that cannot be derived automatically.

Staging can also be applied to statically express and check semantics. The genericity of the suggested solution leaves the choice of type system open, resorting to the C++ ways of typing algorithms only by default.

5.2 Limitations

Like all C++ template meta-programming methods, the solution suffers from the limitations of the template compilation model and from the costs of compilation. Only software of a limited extent can be completely expressed using template meta-programming. Yet, we claim that it suffices to provide the suggested generic implementations to the (computationally intensive) kernels of an application, and to rely on traditional programming models for large-scale software design.

C++ templates are also the reason why error checking and debugging support are still hard to provide for these methods. At the minimum, the library designer is required to obey programming discipline and to have a certain familiarity with library internals to interpret error messages.

The currently biggest limitation, however, is the experimental state of the library, which is not sufficiently stable yet for use in production code. The library still lacks a reasonably-sized code base and, so far, contains only elementary algorithm implementations. In particular, support for the definition of user-defined data structures, i.e., classes, is missing. Such definitions are not trivial if member functions are meant to interact seamlessly with the presented method of generic algorithm implementation.

5.3 Future Work

Naturally, we plan extending the selection of algorithm implementations, not at last to evaluate and improve the library itself.

Furthermore, support for the definition of data structures is needed, which, in turn, will serve the implementation of “interesting” algorithms.

On the other hand, a simplification and abstraction of notion is desirable, in order to express the constructs by as few references to C++ as possible. A simplified notation would help to compare and
combine the solution with other approaches and ways of reasoning, and to possibly port the library to other languages.

References


