Efficient Run-Time Dispatching in Generic Programming with Minimal Code Bloat

Lubomir Bourdev
Adobe Systems Inc.
lbourdev@adobe.com

Jaakko Järvi
Texas A&M University
jarvi@cs.tamu.edu

Abstract

Generic programming using C++ results in code that is efficient but inflexible. The inflexibility arises, because the exact types of inputs to generic functions must be known at compile time. We show how to achieve run-time polymorphism without compromising performance by instantiating the generic algorithm with a comprehensive set of possible parameter types, and choosing the appropriate instantiation at run time. The major drawback of this approach is excessive template bloat, generating a large number of instantiations, many of which are identical at the assembly level. We show practical examples in which this approach quickly reaches the limits of the compiler. Consequently, we combine the method of run-time polymorphism for generic programming with a strategy for reducing the amount of necessary template instantiations. We report on using our approach in GIL, Adobe's open source Generic Image Library. We observed notable reduction, up to 70% at times, in executable sizes of our test programs. Even with compilers that perform aggressive template hoisting at the compiler level, we achieve notable code size reduction, due to significantly smaller dispatching code. The framework draws from both the generic programming and generative programming paradigm, using static metaprogramming to fine tune the compilation of a generic library. Our test bed, GIL, is deployed in a real world industrial setting, where code size is often an important factor.

Categories and Subject Descriptors D.3.3 [*Programming Techniques*]: Language Constructs and Features—Abstract data types; D.3.3 [*Programming Techniques*]: Language Constructs and Features—Polymorphism; D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries

General Terms Design, Performance, Languages

Keywords generic programming, C++ templates, template bloat, template metaprogramming

1. Introduction

Generic programming, pioneered by Musser and Stepanov [19], and introduced to C++ with the STL [24], aims at expressing algorithms at an abstract level, such that the algorithms apply to as broad class of data types as possible. A key idea of generic programming is that this abstraction should incur no performance degradation: once a generic algorithm is specialized for some concrete data types, its performance should not differ from a similar algorithm written directly for those data types. This principle is often referred to as *zero abstraction penalty*. The paradigm of generic programming has been successfully applied in C++, evidenced, e.g., by the STL, the Boost Graph Library (BGL) [21], and many other generic libraries [3, 5, 11, 20, 22, 23]. One factor contributing to this success is the compilation model of templates, where specialized code is generated for every different instance of a template. We refer to this compilation model as the *instantiation model*.

We note that the instantiation model is not the only mechanism for compiling generic definitions. For example, in Java [13] and Eiffel [10] a generic definition is compiled to a single piece of byte or native code, used by all instantiations of the generic definition. C# [9, 18] and the ECMA .NET framework delay the instantiation of generics until run time. Such alternative compilation models address the code bloat issue, but may be less efficient or may require run-time compilation. They are not discussed in this paper.

With the instantiation model, zero abstraction penalty is an attainable goal: later phases of the compilation process make no distinction between code generated from a template instantiation and non-template code written directly by the programmer. Thus, function calls can be resolved statically, which enables inlining and other optimizations for generic code. The instantiation model, however, has other less desirable characteristics, which we focus on in this paper.

In many applications the exact types of objects to be passed to generic algorithms are not known at compile time. In C++ all template instantiations and code generation that they trigger occur at compile time—*dynamic dispatching* to templated functions is not (directly) supported. For efficiency, however, it may be crucial to use an algorithm instantiated for particular concrete types.

In this paper, we describe how to instantiate a generic algorithm with all possible types it may be called with, and generate code that dispatches at run time to the right instantiation. With this approach, we can combine the flexibility of dynamic dispatching and performance typical for the instantiation model: the dispatching occurs only once per call to a generic algorithm, and has thus a negligible cost, whereas the individual instantiations of the algorithms are compiled and fully optimized knowing their concrete input types. This solution, however, leads easily to excessive number of template instantiations, a problem known as *code bloat* or *template bloat*. In the instantiation model, the combined size of the instantiations grows with the number of instantiations: there is typically no code sharing between instantiations of the same templates with different types, regardless of how similar the generated code is.[1]

LCSD '06 October 22nd, Portland, Oregon.
ACM [to be supplied].

[1] At least one compiler, Visual Studio 8, has advanced heuristics that can optimize for code bloat by reusing the body of assembly-level identical

This paper reports on experiences of using the generic programming paradigm in the development of the Generic Image Library (GIL) [5] in the Adobe Source Libraries [1]. GIL supports several image formats, each represented internally with a distinct type. The static type of an image manipulated by an application using GIL is often not known; the type assigned to an image may, e.g., depend on the format it was stored on the disk. Thus, the case described above manifests in GIL: an application using GIL must instantiate the relevant generic functions for all possible image types and arrange that the correct instantiations are selected based on the arguments' dynamic types when calling these functions. Following this strategy blindly may lead to unmanageable code bloat. In particular, the set of instantiations increases exponentially with the number of image type parameters that can be varied independently in an algorithm. Our experience shows that the number of template instantiations is an important design criterion in developing generic libraries.

We describe the techniques and the design we use in GIL to ensure that specialized code for all performance critical program parts is generated, but still keep the number of template instantiations low. Our solution is based on the realization that even though a generic function is instantiated with different type arguments, the generated code is in some cases identical. We describe mechanisms that allow the different instantiations to be replaced with a single common instantiation. The basic idea is to decompose a complex type into a set of orthogonal parameter dimensions (with image types, these include color space, channel depth, and constness) and identify which parameters are important for a given generic algorithm. Dimensions irrelevant for a given operation can be cast to a single "base" parameter value. Note that while this technique is presented as a solution to dealing with code bloat originating from the "dynamic dispatching" we use in GIL, the technique can be used in generic libraries without a dynamic dispatching mechanism as well.

In general, a developer of a software library and the technologies supporting library development are faced with many, possibly competing, challenges, originating from the vastly different context the libraries can be used. Considering GIL, for example, an application such as Adobe Photoshop requires a library flexible enough to handle the variation of image representations at run time, but also places strict constraints on performance. Small memory footprint, however, becomes essential when using GIL as part of a software running on a small device, such as a cellular phone or a PDA. Basic software engineering principles ask for easy extensibility, etc. The design and techniques presented in this paper help in building generic libraries that can combine efficiency, flexibility, extensibility, and compactness.

C++'s template system provides a programmable sub-language for encoding compile-time computations, the uses of which are known as *template metaprogramming* (see e.g. [25], [8, §.10]). This form of generative programming proved to be crucial in our solution: the process of pruning unnecessary instantiations is orchestrated with template metaprograms. In particular, for our metaprogramming needs, we use the Boost Metaprogramming Library (MPL) [2, 14] extensively. In the presentation, we assume some familiarity with the basic principles of template metaprogramming in C++.

The structure of the paper is as follows. Section 2 describes typical approaches to fighting code bloat. Section 3 gives a brief introduction to GIL, and the code bloat problems therein. Section 4 explains the mechanism we use to tackle code bloat, and Section 5 describes how to apply the mechanism with dynamic dispatching to generic algorithms. We report experimental results in Section 6, and conclude in Section 7.

functions. In the results section we demonstrate that our method can result in noticeable code size reduction even in the presence of such heuristics.

2. Background

One common strategy to reduce code bloat associated with the instantiation model is *template hoisting* (see e.g. [6]). In this approach, a class template is split into a non-generic base class and a generic derived class. Every member function that does not depend on any of the template parameters is moved, hoisted, into the base class; also non-member functions can be defined to operate directly on references or pointers to objects of the base-class type. As a result, the amount of code that must be generated for each different instantiation of the derived class decreases. For example, red-black trees are used in the implementation of *associative containers* `map`, `multimap`, `set`, and `multiset` in the C++ Standard Library [15]. Because the tree balancing code does not need to depend on the types of the elements contained in these containers, a high-quality implementation is expected to hoist this functionality to non-generic functions. The GNU Standard C++ Library v3 does exactly this: the tree balancing functions operate on pointers to a non-generic base class of the tree's node type.

In the case of associative containers, the tree node type is split into a generic and non-generic part. It is in principle possible to split a template class into several layers of base classes, such that each layer reduces the number of template parameters. Each layer then potentially has less type variability than its subclasses, and thus two different instantiations of the most derived class may coalesce to a common instantiation of a base class. Such designs seem to be rare.

Template hoisting within a class hierarchy is a useful technique, but it allows only a single way of splitting a data type into sub-parts. Different generic algorithms are generally concerned with different aspects of a data-type. Splitting a data type in a certain way may suit one algorithm, but will be of no help for reducing instantiations of other algorithms. In the framework discussed in this paper, the library developer, possibly also the client of a library, can define a partitioning of data-types, where a particular algorithm needs to be instantiated only with one representative of each equivalence class in the partition.

We define the partition such that differences between types that do not affect the operation of an algorithm are ignored. One common example is pointers - for some algorithms the pointed type is important, whereas for others it is ok to cast to `void*`. A second example is differences due to constness (consider STL's `iterator` and `const_iterator` concept). The generated code for invoking a *non-modifying* algorithm (one which accepts immutable iterators) with mutable iterators will be identical to the code generated for an invocation with immutable iterator. Some algorithms need to operate bitwise on their data, whereas others depend on the type of data. For example, assignment between a pair of pixels is the same regardless of whether they are CMYK or RGBA pixels, whereas the type of pixel matters to an algorithm that sets the color to white, for example.

3. Generic Image Library

The Generic Image Library (GIL) is Adobe's open source image processing library [5]. GIL addresses a fundamental problem in image processing projects — operations applied to images (such as copying, comparing, or applying a convolution) are logically the same for all image types, but in practice image representations in memory can vary significantly, which often requires providing multiple variations of the same algorithm. GIL is used as the framework for several new features planned for inclusion in the next version of Adobe Photoshop. GIL is also being adopted in several other imaging projects inside Adobe. Our experience with these efforts show

that GIL helps to reduce the size of the core image manipulation source code significantly, as much as 80% in a particular case.

Images are 2D (or more generally, n-dimensional) arrays of pixels. Each pixel encodes the color at the particular point in the image. The color is typically represented as the values of a set of *color channels*, whose interpretation is defined by a *color space*. For example, the color red can be represented as 100% red, 0% green, and 0% blue using the RGB color space. The same color in the CMYK color space can be approximated with 0% cyan, 96% magenta, 90% yellow, and 0% black. Typically all pixels in an image are represented with the same color space.

GIL must support significant variation within image representations. Besides color space, images may vary in the ordering of the channels in memory (RGB vs. BGR), and in the number of bits (depth) of each color channel and its representation (8 bit vs. 32 bit, unsigned char vs. float). Image data may be provided in *interleaved* form (RGBRGBRGB...) or in *planar* form where each color plane is separate in memory (RRR..., GGG... BBB...); some algorithms are more efficient in planar form whereas others perform better in interleaved form. In some image representations each row (or the color planes) may be aligned, in which case a gap of unused bytes may be present at the end of each row. There are representations where pixels are not consecutive in memory, such as a sub-sampled view of another image that only considers every other pixel. The image may represent a rectangular sub-image in another image or an upside-down view of another image, for example. The pixels of the image may require some arbitrary transformation (for example an 8-bit RGB view of 16-bit CMYK data). The image data may not be at all in memory (a virtual image, or an image inside a JPEG file). The image may be synthetic, defined by an arbitrary function (the Mandelbrot set), and so forth.

Note that GIL makes a distinction between *images* and *image views*. Images are containers that own their pixels, views do not. Images can return their associated views and GIL algorithms operate on views. For the purpose of this paper, these differences are not significant, and we use the terms image and image views (or just views) interchangeably.

The exact image representation is irrelevant to many image processing algorithms. To compare two images we need to loop over the pixels and compare them pairwise. To copy one image into another we need to copy every pixel pairwise. To compute the histogram of an image, we need to accumulate the histogram data over all pixels. To exploit these commonalities, GIL follows the generic programming approach, exemplified by the STL, and defines abstract representations of images as *concepts*. In the terminology of generic programming, a concept is the formalization of an abstraction as a set of requirements on a type (or types) [4, 16]. A type that implements the requirements of a concept is said to *model* the concept. Algorithms written in terms of image concepts work for images in any representation that model the necessary concepts. By this means, GIL avoids multiple definitions for the same algorithm that merely accommodate for inessential variation in the image representations.

GIL supports a multitude of image representations, for each of which a distinct typedef is provided. Examples of these types are:

- rgb8_view_t: 8-bit mutable interleaved RGB image
- bgr16c_view_t: 16-bit immutable interleaved BGR image
- cmyk32_planar_view_t: 32-bit mutable planar CMYK image
- lab8c_step_planar_view_t: 8-bit immutable LAB planar image in which the pixels are not consecutive in memory

The actual types associated with these typedefs are somewhat involved and not presented here.

GIL represents color spaces with distinct types. The naming of these types is as expected: rgb_t stands for the RGB color space, cmyk_t for the CMYK color space, and so forth. Channels can be represented in different permutations of the same set of color values. For each set of color values, GIL identifies a single color space as the *primary* color space — its permutations are *derived* color spaces. For example, rgb_t is a primary color space and bgr_t is its *derived* color space.

GIL defines two images to be *compatible* if they have the same set and type of channels. That also implies their color spaces must have the same primary color space. Compatible images may vary any other way - planar vs. interleaved organization, mutability, etc. For example, an 8-bit RGB planar image is compatible with an 8-bit BGR interleaved image. Compatible images may be copied from one another and compared for equality.

3.1 GIL Algorithms

We demonstrate the operation of GIL with a simple algorithm, copy_pixels(), that copies one image view to another. Here is one way to implement it:[2]

```
template <typename View1, typename View2>
void copy_pixels(const View1& src, const View2& dst) {
  std::copy(src.begin(), src.end(), dst.begin());
}
```

A requirement of copy_pixels is that the two image view types be compatible and have the same dimensions, and that the destination be mutable. An attempt to instantiate copy_pixels with incompatible images results in a compile-time error.

Each GIL image type supports the begin() and end() member functions as defined in the STL's Container concept. Thus the body of the algorithm just invokes the copy() algorithm from the C++ standard library. If we expand out the std::copy() function, copy_pixels becomes:

```
template <typename View1, typename View2>
void copy_pixels(const View1& src, const View2& dst) {
  typedef typename View1::iterator src_it = src.begin();
  typedef typename View2::iterator dst_it = dst.begin();
  while (src_it != dst.end()) {
    *dst_it++ = *src_it++;
  }
}
```

Each image type is required to have an associated iterator type that implements iteration over the image's pixels. Furthermore, each pixel type must support assignment. Note that the source and target images can be of different (albeit compatible) types, and thus the assignment may include a (lossless) conversion from one pixel type to another. These elementary operations are implemented differently by different image types. A built-in pointer type can serve as the iterator type of a simple interleaved image[3], whereas in a planar RGB image it may be a bundle of three pointers to the corresponding color planes. The iterator increment operator $++$ for interleaved images may resolve to a pointer increment, for step images to advancing a pointer by a given number of bytes, and for a planar RGB iterator to incrementing three pointers. The dereferencing operator $*$ for simple interleaved images returns a reference type; for planar RGB images it returns a *planar reference proxy* object containing three references to the three channels. For a complex image type, such as one representing an RGB view over CMYK data, the dereferencing operator may perform color conversion.

[2] Note that GIL image views don't own the pixels and don't propagate their constness to the pixels, which explains why we take the destination as a const reference. Mutability is incorporated into the image view type.

[3] Assuming the image has no gap at the end of each row

Due to the instantiation model, the calls to the implementations of the elementary image operations in GIL algorithms can be resolved statically and usually inlined, resulting in an efficient algorithm specialized for the particular image types used. GIL algorithms are targeted to match the performance of code hand-written for a particular image type. Any difference in performance from that of hand-written code is usually due to abstraction penalty, for example, the compiler failing to inline a forwarding function, or failing to pass small objects of user-defined types in registers. Modern compilers exhibit zero abstraction penalty with GIL algorithms in many common uses of the library.

3.2 Dynamic dispatching in GIL

Sometimes the exact image type with which the algorithm is to be called is unknown at compile time. For this purpose, GIL implements the variant template, i.e. a discriminated union type. The implementation is very similar to that of the Boost Variant Library [12]. One difference is that the Boost variant template can be instantiated with an arbitrary number of template arguments, while GIL variant accepts exactly one argument [4]. This argument itself represents a collection of types and it must be a model of the Random Access Sequence concept, defined in MPL. For example, the vector template in MPL models this concept. A variant object instantiated with an MPL vector holds an object whose type can be any one of the types contained in the type vector.

Populating a variant with image types, and instantiating another template in GIL, any_image_view, with the variant, yields a GIL image type that can hold any of the image types in the variant. Note the difference to polymorphism via inheritance and dynamic dispatching: in polymorphism via virtual member functions, the set of virtual member functions, and thus the set of algorithms, is fixed but the set of data types implementing those algorithms is extensible; with variant types, the set of data types is fixed, but there is no limit to the number of algorithms that can be defined for those data types. The following code illustrates the use of the any_image_view type:[5]

```
typedef variant<mpl::vector<rgb8_view_t, bgr16c_view_t,
                            cmyk32_planar_view_t,
                            lab8_step_planar_view_t> > my_views_t;

any_image_view<my_views_t> v1, v2;
jpeg_read_view(file_name1, v1);
jpeg_read_view(file_name2, v2);
  ...
copy_pixels(v1, v2);
```

Compiling the call to copy_pixels involves examining the run time types of v1 and v2 and dispatching to the instantiation of copy_pixels generated for those types. Indeed, GIL overloads algorithms for any_image_view types, which do exactly this. Consequently, all run time dispatching occurs at a higher level, rather than at the inner loops of the algorithms; any_image_view containers are practically as efficient as if the exact image type was known at compile time. Obviously, the precondition to dispatching to a specific instantiation is that the instantiation has been generated. Unless we are careful, this may lead to significant template bloat, as illustrated in the next section.

3.3 Template bloat originating from GIL's dynamic dispatching

To ease the definition of lists of types for the any_image_view template, GIL implements type generators. One of these generators is cross_vector_image_view_types, which generates all image types that are combinations of given sets of color spaces and channels, and the interleaved/planar and step/no step policies, as the following example demonstrates:

```
typedef mpl::vector<rgb_t,bgr_t,lab_t,cmyk_t>::type ColorSpaceV;
typedef mpl::vector<bits8,bits16,bits32>::type ChannelV;

typedef any_image_view<cross_vector_image_view_types<
  ColorSpaceV, ChannelV,
  kInterleavedAndPlanar, kNonStepAndStep> > any_view_t;

any_view_t v1, v2;

v1 = rgb8_planar_view_t(..);
v2 = bgr8_view_t(..);

copy_pixels(v1, v2);
```

This code defines any_image_t to be one of $4 \times 3 \times 2 \times 2 = 48$ possible image types. It can have any of the four listed color spaces, any of the three listed channel depths, it can be interleaved or planar and its pixels can be adjacent or non-adjacent in memory. The above code generates $48 \times 48 = 2304$ instantiations. Without any special handling, the code bloat will be out of control.

In practice, the majority of these combinations are between incompatible images, which in the case of run-time instantiated images results in throwing an exception. Nevertheless, such exhaustive code generation is wasteful since many of the cases generate essentially identical code. For example, copying two 8-bit interleaved RGB images or two 8-bit interleaved LAB images (with the same channel types) results in the same assembly code — the interpretation of the channels is irrelevant for the copy operation. The following section describes how we can use metaprograms to avoid generating such identical instantiations.

4. Reducing the Number of Instantiations

Our strategy for reducing the number of instantiations is based on decomposing a complex type into a set of orthogonal parameter dimensions (such as color space, channel depth, constness) and identifying which dimensions are important for a given operation. Dimensions irrelevant for a given operation can be cast to a single "base" parameter value. For example, for the purpose of copying, all LAB and RGB images could be treated as RGB images. As mentioned in Section 2, for each algorithm we define a partition among the data types, select the equivalence class representatives, and only generate an instance of the algorithm for these representatives. We call this process *type reduction*.

Type reduction is implemented with metafunctions which map a given data type and a particular algorithm to the class representative of that data type for the given algorithm. By default, that reduction is identity:

```
template <typename Op, typename T>
struct reduce { typedef T type; };
```

By providing template specializations of the reduce template for specific types, the library author can define the partition of types for each algorithm. We return to this point later. Note that the algorithm is represented with the type Op here; we implement GIL algorithms internally as function objects instead of free-standing function templates. One advantage is that we can represent the algorithm with a template parameter.

We need a generic way of invoking an algorithm which will apply the reduce metafunction to perform type reduction on its arguments prior to entering the body of the algorithm. For this purpose, we define the apply_operation function[6]:

[4] The Boost Variant Library offers similar functionality with the make_variant_over metafunction.

[5] The mpl::vector instantiation is a compile-time data structure, a vector whose elements are types; in this case the four image view types.

[6] Note that reinterpret_cast is not portable. To cast between two arbitrary types GIL uses instead static_cast<T*>(static_cast<void*>(arg)). We omit this detail for readability.

```
struct invert_pixels_op {
  typedef void result_type;
  template <typename View>
  void operator()(const View& v) const {
    const int N = View::num_channels;
    typename View::iterator it = v.begin();
    while (it != v.end()) {
      typename View::reference pix=*it;
      for (int i=0; i<N; ++i)
        pix[i]=invert_channel(pix[i]);
      ++it;
    }
  }
};
template <typename View>
inline void invert_pixels(const View& v) {
  apply_operation(v, invert_pixels_op());
}
```

Figure 1. The invert_pixels algorithm.

```
template <typename Arg, typename Op>
inline typename Op::result_type
apply_operation(const Arg& arg, Op op) {
    typedef typename reduce<Op,Arg>::type base_t;
    return op(reinterpret_cast<const base_t&>(arg));
}
```

This function provides the glue between our technique and the algorithm. We have overloads for the one and two argument cases, and overloads for variant types. The apply_operation function serves two purposes — it applies reduction to the arguments and invokes the associated function. As the example above illustrates, for templated types the second step amounts to a simple function call. In Section 5 we will see that for variants this second step also resolves the static types of the objects stored in the variants, by going through a switch statement.

Let us consider an example algorithm, invert_pixels. It inverts each channel of each pixel in an image. Figure 1 shows a possible implementation (which ignores performance and focuses on simplicity) that can be invoked via apply_operation.

With the definitions this far, nothing has changed from the perspective of the library's client. The invert_pixels() function merely forwards its parameter to apply_operation(), which again forwards to invert_pixels_op(). Both apply_operation() and invert_pixels() are inlined, and the end result is the same as if the algorithm implementation was written directly in the body of invert_pixels(). With this arrangement, however, we can control instantiations with defining specializations for the reduce metafunction. For example, the following statement will cause 8-bit LAB images to be reduced to 8-bit RGB images when calling invert_pixels:

```
template<>
struct reduce<invert_pixels_op, lab8_view_t> {
  typedef rgb8_view_t type;
};
```

This approach extends to algorithms taking more than one argument — all arguments can be represented jointly as a tuple. The reduce metafunction for binary algorithms can have specializations for std::pair of any two image types the algorithm can be called with — Section 4.1 shows an example. Each possible pair of input types, however, can be a large space to consider. In particular, using variant types as arguments to binary algorithms (see Section 5) generates a large number of such pair types, which can take a toll on compile times. Fortunately, for many binary algorithms it is possible to apply unary reduction independently on each of the input arguments first and only consider pairs of the argument types after reduction – this is potentially a much smaller set of pairs. We call such preliminary unary reduction *pre-reduction*. Here is the apply_operation taking two arguments:

```
template <typename Arg1 typename Arg2, typename Op>
inline typename Op::result_type
apply_operation(const Arg1& arg1, const Arg2& arg2, Op op) {
    // unary pre−reduction
    typedef typename reduce<Op,Arg1>::type base1_t;
    typedef typename reduce<Op,Arg2>::type base2_t;

    // binary reduction
    typedef std::pair<const base1_t*, const base2_t*> pair_t;
    typedef typename reduce<Op,pair_t>::type base_pair_t;

    std::pair<const void*,const void*> p(&arg1,&arg2);
    return op(reinterpret_cast<const base_pair_t&>(p));
}
```

As a concrete example of a binary algorithm that can be invoked via apply_operation, the copy_pixels() function can be defined as follows:

```
struct copy_pixels_op {
  typedef void result_type;
  template <typename View1, typename View2>
  void operator()(const std::pair<const View1*,
                                  const View2*>& p) const {
    typedef typename View1::iterator src_it = p.first→ begin();
    typedef typename View2::iterator dst_it = p.second→ begin();
    while (src_it != dst.end()) {
      *dst_it++ = *src_it++;
    }
  }
};

template <typename View1, typename View2> inline void
copy_pixels(const View1& src, const View2& dst) {
    apply_operation(src, dst, copy_pixels_op());
}
```

We note that the type reduction mechanism relies on an unsafe cast operation, which relies on programmers assumptions not checked by the compiler or the run time system. The library author defining the reduce metafunction must thus know the implementation details of the types that are being mapped to the class representative, as well as the implementation details of the class representative. A client of the library defining new image types can specialize the reduce template to specify a partition within those types, without needing to understand the implementations of the existing image types in the library.

4.1 Defining reduction functions

In general, the reduce metafunction can be implemented by whatever means is most suitable, most straightforwardly by enumerating all cases separately. Commonly a more concise definition is possible. Also, we can identify "helper" metafunctions that can be reused in the type reduction for many algorithms. To demonstrate, we describe our implementation for the type reduction of the copy_pixels algorithm. Even though we use MPL in GIL extensively, following the definitions requires no knowledge of MPL; here we use a traditional static metaprogramming style of C++, where branching is expressed with partial specializations.

The copy_pixels algorithm operates on two images — we thus apply the two phase reduction strategy discussed in Section 4, first pre-reducing each image independently, followed by the pair-wise reduction.

To define the type reductions for GIL image types, reduce must be specialized for them:

```
template <typename Op, typename L>
struct reduce<Op, image_view<L> >
  : public reduce_view_basic<Op, image_view<L>,
       view_is_basic<image_view<L> >::value> {};

template <typename Op, typename L1, typename L2>
struct reduce<Op, std::pair<const image_view<L1>*,
                            const image_view<L2>*> >
  : public reduce_views_basic<
      Op, image_view<L1>, image_view<L2>,
      mpl::and_<view_is_basic<image_view<L1> >,
                view_is_basic<image_view<L2> > >::value> {};
```

Note the use the use *metafunction forwarding* idiom from the MPL, where one metafunction is defined in terms of another metafunction by inheriting from it, here reduce is defined in terms of reduce_view_basic.

The first of the above specializations will match any GIL image_view type, the second any pair[7] of GIL image_view types. These specializations merely forward to reduce_view_basic and reduce_views_basic—two metafunctions specific to reducing GIL's image view types. view_is_basic template defines a compile time predicate that tests whether a given view type is one of GIL's built-in view types, rather than a view type defined by the client of the library. We can only define the reductions of view types known to the library, the ones satisfying the prediacte—for all other types GIL applies identity mappings using the following default definitions for reduce_view_basic and reduce_views_basic:

```
template <typename Op, typename View, bool IsBasic>
struct reduce_view_basic { typedef View type; };

template <typename Op, typename V1, typename V2,
          bool AreBasic>
struct reduce_views_basic {
  typedef std::pair<const V1*, const V2*> type;
};
```

The above metafunctions are not specific to a particular type reduction and are shared by reductions of all algorithms.

The following reductions that operate on the level of color spaces are also useful for many algorithms in GIL. Different color spaces with the same number of channels can all be reduced to one common type. We choose rgb_t and rgba_t as the class representatives for three and four channel color spaces, respectively. Note that we do not reduce different permutations of channels. For example, we cannot reduce bgr_t to rgb_t because that will violate the channel ordering.

```
template <typename Cs> struct reduce_color_space {
  typedef Cs type;
};
template <> struct reduce_color_space<lab_t> {
  typedef rgb_t type;
};
template <> struct reduce_color_space<hsb_t> {
  typedef rgb_t type;
};
template <> struct reduce_color_space<cmyk_t> {
  typedef rgba_t type;
};
```

We can similarly define a binary color space reduction — a metafunction that takes a pair of (compatible) color spaces and returns a pair of reduced color spaces. For brevity, we only show the interface of the metafunction:

[7] We represent the two types as a pair of constant pointers because it makes the implementation of reduction with a variant (described in Section 5) easier.

```
template <typename SrcCs, typename DstCs>
struct reduce_color_spaces {
    typedef ... first_t;
    typedef ... second_t;
};
```

The equivalence classes defined by this metafunction represent the color space pairs where the mapping of channels from first to second color space is preserved. We can represent such mappings with a tuple of integers. For example, the mapping of pair<rgb_t,bgr_t> is $\langle 2, 1, 0\rangle$, as the first channel r maps from the position 0 to position 2, g from position 1 to 1, and b from 2 to 1. Mappings for pair<bgr_t,bgr_t> and pair<lab_t,lab_t> are represented with the tuple $\langle 0, 1, 2\rangle$. We have identified eight mappings that can represent all pairs of color spaces that are used in practice. New mappings can be introduced when needed as specializations.

With the above helper metafunctions, we can now define the type reduction for copy_pixels. First we define the unary pre-reduction that is performed for each image view type independently. We perform reduction in two aspects of the image: the color space is reduced with the reduce_color_space helper metafunction, and both mutable and immutable views are unified. We use GIL's derived_view_type metafunction (we omit the definition for brevity) that takes a source image view type and returns a related image view in which some of the parameters are different. In this case we are changing the color space and mutability:

```
template <typename View>
struct reduce_view_basic<copy_pixels_fn,View,true> {
private:
  typedef typename
    reduce_color_space<typename View::color_space_t>::type Cs;
public:
  typedef typename derived_view_type<
    View, use_default, Cs, use_default, use_default, mpl::true_
  >::type type;
};
```

Note that this reduction introduces a slight problem — it would allow us to copy (incorrectly) between some incompatible images — for example from hsb8_view_t into lab8_view_t, as they both will be reduced to rgb8_view_t. However, such calls should never occur, as calling copy_pixels with incompatible images violates its precondition. Even though this pre-reduce significantly improves compile times, due to the above objection we did not use it in our measured experiments.

The first step of binary reduction is to check whether the two images are compatible; the views_are_compatible predicate provides this information. If the images are not compatible, we reduce to error_t — a special tag denoting type mismatch error. All algorithms throw an exception when given error_t:

```
template <typename V1, typename V2>
struct reduce_views_basic<copy_pixels_fn, V1, V2, true>
   : public reduce_copy_pixop_compat<V1,V2,
               mpl::and_<views_are_compatible<V1,V2>,
               view_is_mutable<V2> >::value > {};

template <typename V1, typename V2, bool IsCompatible>
struct reduce_copy_pixop_compat {
  typedef error_t type;
};
```

Finally, if the two image views are compatible, we reduce their color spaces pairwise, using the reduce_color_spaces metafunction discussed above. Figure 2 shows the code, where the metafunction derived_view_type again generates the reduced view types that change the color spaces, but keep other aspects of the image view types the same.

Note that we can easily reuse the type reduction policy for copy_pixels for other algorithms for which the same policy applies:

```
template <typename V1, typename V2>
struct reduce_copy_pixop_compat<V1, V2, true> {
private:
  typedef typename V1::color_space_t Cs1;
  typedef typename V2::color_space_t Cs2;
  typedef typename
    reduce_color_spaces<Cs1,Cs2>::first_t RCs1;
  typedef typename
    reduce_color_spaces<Cs1,Cs2>::second_t RCs2;
  typedef typename
    derived_view_type<V1, use_default, RCs1>::type RV1;
  typedef typename
    derived_view_type<V2, use_default, RCs2>::type RV2;
public:
  typedef std::pair<const RV1*, const RV2*> type;
};
```

Figure 2. Type reduction for copy_pixels of compatible images.

```
template <typename V, bool IsBasic>
struct reduce_view_basic<resample_view_fn, V, IsBasic>
  : public reduce_view_basic<copy_pixels_fn, V, IsBasic> {};

template <typename V1, typename V2, bool AreBasic>
struct reduce_views_basic<resample_view_fn, V1, V2, AreBasic>
  : public reduce_views_basic<copy_pixels_fn, V1, V2, AreBasic> {};
```

5. Minimizing Instantiations with Variants

Type reduction is most necessary, and most effective with variant types, such as GIL-s any_image_view, as a single invocation of a generic algorithm would normally require instantiations to be generated for all types in the variant, or even for all combinations of types drawn from several variant types. This section describes how we apply the type reduction machinery in the case of variant types.

Variants are comprised of three elements — a type vector of possible types the variant can store (Types), a run-time value (index) to this vector indicating the type of the object currently stored in the variant, and the memory block containing the instantiated object (bits). Invoking an algorithm, which we represent as a function object, amounts to a switch statement over the value of index, each case N of which casts bits to the N-th element of Types and passes the casted value to the function object. We capture this functionality in the apply_operation_base template:[8]

```
template <typename Types, typename Bits, typename Op>
typename Op::result_type
apply_operation_base(const Bits& bits, int index, Op op) {
    switch (index) {
    ...
    case N: return op(reinterpret_cast<const
                typename mpl::at_c<Types, N>::type&>(bits));
    ...
    }
}
```

As we discussed before, such code instantiates the algorithm with every possible type and can lead to code bloat. Instead of calling this function directly from the apply_operation function template overloaded for variants, we first subject the Types vector to reduction:

[8] The number of cases in the switch statement equals the size of the Types vector. We use the preprocessor to generate such functions with different number of case statements and we use specialization to select the correct one at compile time.

```
template <typename Types, typename Op>
struct unary_reduce {
    typedef ... reduced_t;
    typedef ... unique_t;
    typedef ... indices_t;
    static int map_index(int index) {
        return dynamic_at_c<indices_t>(index);
    }
    template <typename Bits>
    static typename Op::result_type
    apply(const Bits& bits, int index, Op op) {
        return apply_operation_base<unique_t>
            (bits,map_index(index),op);
    }
}
```

Figure 3. Unary reduction for variant types.

```
template <typename Types, typename Op>
inline typename Op::result_type
apply_operation(const variant<Types>& arg, OP op) {
  return unary_reduce<Types,Op>::
    template apply(arg._bits,arg._index,op);
}
```

The unary_reduce template performs type reduction, and its apply member function invokes apply_operation_base with the smaller, reduced, set of types. The definition of unary_reduce is shown in Figure 3. The definitions of the three typedefs are omitted, but they are computed as follows:

- reduced_t — a type vector that holds the reduced types corresponding to each element of Types. That is, reduced_t[i] == reduce<Op, Types[i]>::type
- unique_t — a type set containing the same elements as the type vector reduced_t, but without duplicates.
- indices_t — a type set containing the indices (represented as MPL integral types, which wrap integral constants into types) mapping the reduced_t vector onto the unique_t set, i.e., reduced_t[i] == unique_t[indices_t[i]]

The dynamic_at_c function is parameterized with a type vector of MPL integral types, which are wrappers that represent integral constants as types. The dynamic_at_c function takes an index to the type vector and returns the element in the type vector as a run-time value. That is, we are using a run-time index to get a run-time value out from a type vector. The definitions of dynamic_at_c function are generated with the preprocessor; the code looks similar to the following[9]:

```
template <typename Ints>
static int dynamic_at_c(int index) {
   static int table[] = {
      mpl::at_c<Ints,0>::value,
      mpl::at_c<Ints,1>::value,
      ...
   };
   return table[index];
}
```

Some algorithms, like copy_pixels, may have two arguments each of which may be a variant. Without any type reduction, applying a

[9] In reality the number of table entries must equal the size of the type vector. We use the Boost Preprocessor Library [17] to generate function objects specialized over the size of the type vector, whose application operators generate tables of appropriate sizes and perform the lookup. We dispatch to the right specialization at compile time, thereby assuring the most compact table is generated.

binary variant operation is implemented using a double-dispatch — we first invoke apply_operation_base with the first variant, passing it a function object, which, when invoked, will in turn call apply_operation_base on the second argument, passing it the original function. If N is the number of types in each input variant, this implementation will generate N^2 instantiations of the algorithm and $N + 1$ switch statements having N cases each.

We can, however, possibly achieve more reduction if we consider the argument types together, rather than each independently. Figure 4 shows the definition of the overload for the binary apply_operation function template. We leave several details without discussion, but the general strategy can be observed from the code:

1. Perform unary_reduce on each input argument to obtain the set of unique reduced types, unique1_t and unique2_t. A binary algorithm can define pre-reductions for its argument types, such as the color space reductions described in Section 4.1. Any pre-reductions at this step are beneficial, as they reduce the amount of compile-time computations preformed in the next step.
2. Compute bin_types, a type vector for the cross-product of the unique pre-reduced types. Its elements are all possible types of the form std::pair<const T1*, const T2*> with T1 and T2 drawn from unique1_t and unique2_t respectively.
3. Perform unary reduction on bin_types, to obtain unique_t — the set of unique pairs after reducing each pair under the binary operation.

Finally, to invoke the binary operation we use a switch statement over the unique pairs of types left over after reduction. We map the two indices to the corresponding single index over the unique set of pairs. This version is advantageous because it instantiates far fewer than N^2 number of types and uses a single switch statement instead of two nested ones.

```
template <typename Types1, typename Types2, typename Op>
struct binary_reduce {
  typedef unary_reduce<Types1,Op> unary1_t;
  typedef unary_reduce<Types2,Op> unary2_t;
  typedef typename unary1_t::unique_t unique1_t;
  typedef typename unary2_t::unique_t unique2_t;

  typedef cross_product_pairs<unique1_t, unique2_t> bin_types;
  typedef unary_reduce<bin_types,Op> binary_t;
  typedef typename binary_t::unique_t unique_t;

  static inline int map_indices(int index1, int index2) {
    int r1=unary1_t::map_index(index1);
    int r2=unary1_t::map_index(index2);
    return bin_reduced_t::map_index(
                        r2*mpl::size<unique1_t>::value + r1);
  }
public:
  template <typename Bits1, typename Bits2>
  static typename Op::result_type
  apply(const Bits1& bits1, int index1,
        const Bits2& bits2, int index2, Op op) {
    std::pair<const void*,const void*> pr(&bits1, &bits2);

    return apply_operation_base<unique_t>
              (pr, map_indices(index1,index2),op);
  }
};

template <typename T1, typename T2, typename BinOp>
inline typename BinOp::result_type apply_operation(
  const variant<T1>& arg1, const variant<T2>& arg2, BinOp op)
{
  return binary_reduce<T1,T2,Op>::
      template apply(arg1._bits,arg1._index,
                     arg2._bits,arg2._index, op);
}
```

Figure 4. Binary reduction for variant types.

6. Experimental Results

To assess the effectiveness of type reduction in practice, we measured the executable sizes, and compilation times, of programs that called GIL algorithms with objects of variant types when type reduction was applied, and when it was not applied.

6.1 Compiler Settings

For our experiments we used the C++ compilers of GCC 4.0 on OS X 10.4 and Visual Studio 8 on Windows XP. For GCC we used the optimization flag $-$O2, and removed the symbol information from the executables with the Unix *strip* command prior to measuring their size. Visual Studio 8 was set to compile in release mode, using all settings that can help reduce code size, in particular the "Minimize Size" optimization (/O1), link-time code generation (/Gl), and eliminating unreferenced data (/OPT:REF). With these the compiler can in some cases detect that two different instances of template functions generate the same code, and avoid the duplication of that code. This makes template bloat a lesser problem in the Visual Studio compiler, as type reduction possibly occurs directly in the compiler. We show, however, improvement even with the most aggressive code-size minimization settings.

6.2 Test Images

For testing type reduction with unary operations, we use an extensive variant of GIL image views, varying in color space (Grayscale, RGB, BGR, LAB, HSB, CMYK, RGBA, ABGR, BGRA, ARGB), in channel depth (8-bit, 16-bit and 32-bit) and in whether the pixels are consecutive in memory or offset by a run-time specified step. This amounts to $10 \times 3 \times 2 = 60$ combinations of interleaved images. In addition, we include planar versions for the primary color spaces (RGB, LAB, HSB, CMYK and RGBA) which adds another $5 \times 3 \times 2 = 30$ combinations for a total of 90 image types.[10]

Binary operations result in explosion in the number of combinations to consider for type reduction. The practical upper limit for direct reduction, with today's compilers and typical desktop computers, is about 20×20 combinations; much beyond that consumes notable amounts of compilation resources.[11] Thus, for binary operations we use two smaller test sets. Test B consists of ten images — Grayscale, BGR, RGB, step RGB, planar RGB, planar step RGB, LAB, step LAB, planar LAB, planar step LAB, all of which are in 8-bit. Test C consists of twelve 8-bit images — in RGB, LAB and HSB, each of which can be planar or interleaved, step or non-step.

To summarize: the test set A contains 90 image types, B contains 10 image types, and C contains 12 image types.

6.3 Test Algorithms

We tested with three algorithms — invert_pixels, copy_pixels and resample_view.

[10] We split the images in two sets because GIL does not allow planar versions of grayscale (as it is identical to interleaved) or derived color spaces (because they can be represented by the primary color spaces by rearranging the order of the pointers to the color planes in the image construction).

[11] GIL determines how complex a given binary type reduction will be and suppresses computing it directly when the number of combinations exceeds a limit. In such a case, the binary operation is represented via double-dispatch as two nested unary operations. This allows more complex binary functions to compile, but the type reduction may miss some possibilities for sharing instantiations.

	S_n	S_r	Decrease in %
Test 1.	201.6	107.5	47%
Test 2.	252.8	75.9	70%
Test 3.	259.8	144.0	45%
Test 4.	318.7	98.8	69%
Test 5.	62.2	31.2	50%

Table 1. Size, in kilobytes, of the generated executable in the five test programs compiled with GCC 4.0 C++ compiler, without (S_n) and with (S_r) type reduction. The fourth column shows the percent decrease in the size of the generated code that was achieved with type reduction.

	S_n	S_r	Decrease in %
Test 1.	42.0	34.5	18%
Test 2.	41.5	26.0	37%
Test 3.	46.0	42.0	8%
Test 4.	33.5	34.0	-1%
Test 5.	24.0	16.5	31%

Table 2. Size, in kilobytes, of the generated executable in the five test programs compiled with Visual Studio 8's C++ compiler, without (S_n) and with (S_r) type reduction. The fourth column shows the percent decrease in the size of the generated code that was achieved with type reduction.

The unary algorithm invert_pixels inverts each channel of each pixel in an image. Although less useful than other algorithms, invert_pixels is simple and allows us to measure the effect of our technique without introducing too much GIL-related code. As a channel-independent operation, invert_pixels does not depend on the color space or ordering of the channels. We tested invert_pixels with the test set A: type reduction maps the 90 image types in test set A down to 30 equivalence classes.

The copy_pixels algorithm, as discussed in Sections 3 and 4, is a binary algorithm performing channel-wise copy between compatible images and throws an exception when invoked with incompatible images. Applied to test images B, our reduction for copy_pixels reduces the image pair types from $10 \times 10 = 100$ down to 26 (25 plus one "incompatible image" case). Without this reduction there are 42 compatible combinations and 58 incompatible ones. The code for the invalid combinations is likely to be shared even without reduction. Thus our reduction transforms 43 cases into 26 cases, which is approximately a 40% reduction.

For test images C, our reduction for copy_pixels reduces the image pairs from $12 \times 12 = 144$ down to 17 (16 plus the "incompatible image" case). Without the reduction, there would be 48 valid and 96 invalid combinations. Thus our reduction transforms 49 into 17 cases, which is approximately a 65% reduction.

We also use another binary operation — resample_view. It resamples the destination image from the source under an arbitrary geometric transformation and interpolates the results using bicubic, bilinear or nearest-neighbor methods. It is a bit more involved than copy_pixels and is therefore less likely to be inlined. It shares the same reduction rules as copy_pixels (works for compatible images and throws an exception for incompatible ones). We test resample_pixels with test images B and C (again, A is too big for a binary algorithm to handle).

In summary we are running 5 tests: (1) copy_pixels on test images B, (2) copy_pixels on test images C, (3) resample_view on test images B, (4) resample_view on test images C, and (5) invert_pixels on test images A.

6.4 Test Results

Our results are obtained as follows: For each of the five tests in an otherwise empty program, we construct an instance of any_image with the corresponding image type set and invoke the corresponding algorithm. We measure the size of the resulting executable and subtract from it the size of the executable if the algorithm is not invoked (but the any_image_view instance is still constructed). The resulting difference in code sizes can thus be attributed to just the code generated from invoking the algorithm. We compute these differences for both platforms, with and without the reduction mechanism, and report the results on Tables 1 and 2.

The results show that we are, on the average, cutting the executable size by more than half under GCC and as much as 70% at times. Since Visual Studio can already avoid generating instantiations whose assembly code is identical, our gain with this compiler is less pronounced. However, we can still observe reduction in the executable size, as much as 32% at times. We believe this is due to two factors — first, Visual Studio's optimization cannot be applied when the code is inlined (which is the case for tests 1, 2 and 5). Indeed those tests show the largest gain. But even for non-inlined code in test 3 we observed a notable reduction. We believe this is due to the simplification of the switch statements. Test 3 without reduction generates 11 (nested) switch statements of 10 cases each, whereas we only generate one switch statement with 26 cases. We also tried inlining resample_view under Visual Studio and got roughly 30% code reduction for tests 3 and 4, (in addition to being about 20% faster to compile, and slightly faster to execute since we avoid two function calls and a double-dispatch).

	Visual Studio 8	GCC
Test 1.	106%	116%
Test 2.	78%	97%
Test 3.	87%	118%
Test 4.	75%	103%
Test 5.	194%	307%

Table 3. The effect of type reduction to compilation times in the five test programs. The percentages are computed as $100 \times T_r/T_n$, where T_n is the compilation time without type reduction and T_r the compilation time using type reduction.

We also measured the time to compile each of the five tests of both platforms when reduction is enabled and compared it to the time when no reduction is enabled. The results are reported in Table 3. We believe there are two main factors in play. On the one hand our reduction techniques involve some heavy-duty template meta-programming, which slows down compiling. On the other hand, the number of instantiated copies of the algorithm is greatly reduced, which reduces the amount of work for the later phases of compiling, in particular if the algorithm's implementation is of substantial size. In addition, a large portion of the types generated during the reduction step are not algorithm-dependent and might be reused when another related algorithm is compiled with the same image set. Finally, when compile times are a concern, our technique may be enabled only towards the end of the product cycle.

7. Conclusions

Combining run-time polymorphism and generic programming with the instantiation model of C++ is non-trivial. We show how variant types can be used for this purpose but, without caution, this easily leads to a severe code bloat. As its main contribution, the paper describes library mechanism for significantly reducing code bloat that results from invoking generic algorithms with variant types, and demonstrates their effectiveness in the context of a production quality generic library.

We discussed the problems of the traditional class-centric approach to addressing code bloat: template hoisting within class hi-

erarchies. This approach requires third-party developers to abide by a specific hierarchy in a given module, and can be inflexible — one hierarchy may allow template hoisting for certain algorithms but not for others. Moreover, complex relationships involving two or more objects may not be representable with a single hierarchy.

We presented an alternative, algorithm-centric approach to addressing code bloat, which allows the definition of partitions among types, each specific to one or more generic algorithms. The algorithms need to be instantiated only for one representative of the equivalence class in each partition. Our technique does not enforce a particular hierarchical structure that extensions to the library must follow. The rules for type reduction are algorithm-dependent and implemented as metafunctions. The clients of the library can define their own equivalence classes by specializing a particular type reduction template defined in a generic library, and have the induced type reductions be applied when using the generic algorithms. Also, new algorithms can be introduced by third-party developers and all they need to do is define the reduction rules for their algorithms. Algorithm reduction rules may be inherited; we discussed the copy_pixels and resample_view algorithms which have identical reduction rules.

The primary disadvantage of our technique is that it relies on a cast operation, the correctness of which is not checked. The reduction specifications declare that a given type can be cast to another given type when used in a given algorithm. That requires intimate knowledge of the type and the algorithm. Nevertheless, we believe the generality and effectiveness of algorithm-centric type reduction justify the safety concerns. We demonstrated that this technique can result in reducing the size of the generated code in half for compilers that don't support template bloat reduction. Even for compilers that employ aggressive pruning of duplicate identical template instantiations, our technique can result in further noticeable decrease in code size.

The framework presented in this paper is essentially an *active library*, as defined by Czarnecki et al. [7]. It draws from both generic and generative programming, static metaprogramming with C++ templates in particular. We accomplish a high degree of reuse and good performance with the generic programming approach to library design. Static metaprogramming allows us to fine tune the library's internal implementation — for example, to decrease the amount of code to be generated.

Our future plans include experimenting with the framework in domains other than imaging. We have experience on generic libraries for linear algebra, which seems to be a promising domain, sharing similarities with imaging: a large number of variations in many aspects of the data types (matrix shapes, element types, storage orders, etc.).

Acknowledgments

We are grateful for Hailin Jin for his contributions to GIL and insights on early stages of this work. This work was in part supported by the NSF grant CCF-0541014.

References

[1] *Adobe Source Libraries*, 2006. `opensource.adobe.com`.

[2] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004.

[3] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. STAPL: An adaptive, generic parallel C++ library. In *Languages and Compilers for Parallel Computing*, volume 2624 of *Lecture Notes in Computer Science*, pages 193–208. Springer, August 2001.

[4] Matthew H. Austern. *Generic programming and the STL: Using and extending the C++ Standard Template Library*. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[5] Lubomir Bourdev and Hailin Jin. *Generic Image Library*, 2006. `opensource.adobe.com/gil`.

[6] Martin D. Carroll and Margaret A. Ellis. *Designing and Coding Reusable C++*. Addison-Wesley, 1995.

[7] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glck, David Vandevoorde, and Todd Veldhuizen. Generative programming and active libraries (extended abstract). In M. Jazayeri, D. Musser, and R. Loos, editors, *Generic Programming. Proceedings*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39. Springer-Verlag, 2000.

[8] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming Methods, Tools, and Applications*. Addison-Wesley, 2000.

[9] ECMA. *C# Language Specification*, June 2005. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`.

[10] ECMA International. *Standard ECMA-367: Eiffel analysis, design and programming Language*, June 2005.

[11] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, a computational geometry algorithms library. *Software – Practice and Experience*, 30(11):1167–1202, 2000. Special Issue on Discrete Algorithm Engineering.

[12] Eric Friedman and Itay Maman. The Boost.Variant library. `http://www.boost.org/libs/variant`, January 2004.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[14] Aleksei Gurtovoy and David Abrahams. The Boost C++ metaprogramming library. `www.boost.org/libs/mpl`, 2002.

[15] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. Geneva, Switzerland, 1998.

[16] D. Kapur and D. Musser. Tecton: a framework for specifying and verifying generic system components. Technical Report RPI–92–20, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180, July 1992.

[17] Vesa Karvonen and Paul Mensonides. The Boost.Preprocessor library. `http://www.boost.org/libs/preprocessor`, 2002.

[18] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET Common Language Runtime. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2001. ACM Press.

[19] David A. Musser and Alexander A. Stepanov. Generic Programming. In *Proceedings of International Symposium on Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 13–25, Rome, Italy, 1988.

[20] W. R. Pitt, M. A. Williams, M. Steven, B. Sweeney, A. J. Bleasby, and D. S. Moss. The Bioinformatics Template Library–generic components for biocomputing. *Bioinformatics*, 17(8):729–737, 2001.

[21] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[22] Jeremy Siek and Andrew Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

[23] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. Generic programming for high performance numerical linear algebra. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[24] A. Stepanov and M. Lee. The Standard Template Library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories, April 1994. `http://www.hpl.hp.com/techreports`.

[25] Todd L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in *C++ Gems*, ed. Stanley Lippman.