An Adaptable and Extensible Geometry Kernel

Susan Hert Michael Hoffmann Lutz Kettner
Sylvain Pion Michael Seel

September 27, 2001

MPI–I–2001–1-004 September 2001

Authors' Addresses

Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
Email: hert@mpi-sb.mpg.de

Institute for Theoretical Computer Science
ETH Zurich
CH-8092 Zurich, Switzerland
Email: hoffmann@inf.ethz.ch

University of North Carolina at Chapel Hill, USA.
Email: kettner@cs.unc.edu

INRIA, Sophia Antipolis - France.
Email: Sylvain.Pion@sophia.inria.fr

Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85 66123 Saarbrücken, Germany
Email: seel@mpi-sb.mpg.de

Abstract

Geometric algorithms are based on geometric objects such as points, lines and circles. The term *kernel* refers to a collection of representations for constant-size geometric objects and operations on these representations. This paper describes how such a geometry kernel can be designed and implemented in C++, having special emphasis on adaptability, extensibility and efficiency. We achieve these goals following the generic programming paradigm and using templates as our tools. These ideas are realized and tested in CGAL [10], the Computational Geometry Algorithms Library.

Keywords

Computational geometry, library design, generic programming.

1 Introduction

Geometric algorithms that manipulate constant-size objects such as circles, lines, and points are usually described independent of any particular representation of the objects. It is assumed that these objects have certain operations defined on them and that simple predicates exist that can be used, for example, to compare two objects or to determine their relative position. Algorithms are described in this way because all representations are equally valid as far as the correctness of an algorithm is concerned. Also, algorithms can be more concisely described and are more easily seen as being applicable in many settings when they are described in this more generic way.

We illustrate here that one can achieve the same advantages when implementing algorithms by encapsulating the representation of objects and the operations and predicates for the objects into a geometry kernel. Algorithms interact with geometric objects only through the operations defined in the kernel. This means that the same implementation of an algorithm can be used with many different representations for the geometric objects. Thus, the representation can be chosen to be the one most applicable (*e.g.*, the most robust or most efficient) for a particular setting.

Regardless of the representation chosen by a particular kernel, it cannot hope to satisfy the needs of every application. For example, for some applications one may wish to maintain additional information with each point during the execution of an algorithm or one may wish to apply a two-dimensional algorithm to a set of coplanar points in three dimensions. Both of these things are easily accomplished if the algorithm in question is implemented in a generic way to interact with objects through a kernel and the kernel is implemented to allow types and operations to be redefined, that is, if the kernel is easily adaptable. It is equally important that a kernel be extensible since some applications may require not simply modifications of existing objects and operations but addition of new ones.

Although adaptability and extensibility are important and worthwhile goals to strive for, one has to keep in mind that the elements of the kernel form the very basic and fundamental building blocks of a geometric algorithm built on top. Hence, we are not willing to accept *any* loss in efficiency on the kernel level. Indeed, using template programming techniques one can achieve genericity without sacrifying runtime-performance by resolving the arising overhead during compile-time.

After discussing previous work on the design of geometry kernels (Section 2), we give a general description of our new kernel concept (Section 3). We then describe how this concept can be realized in an adaptable and extensible way under the generic programming paradigm [24, 25] (Sections 4 through 7). Section 8 illustrates the use of such a kernel and shows how the benefits described above are realized. Finally, we describe the models of this type of kernel that are provided in CGAL (Section 9).

As our implementation is in C++ [9], we assume the reader is somewhat familiar with this language. Stroustrup [30] provides a general introduction to C++ template programming, which is used extensively in our design. Parts of the design of the library were inspired by the STL. Austern [2] provides a good reference for generic programming and the STL, and a good reference for the C++ Standard Library is the book of Josuttis [19].

2 Motivation and Previous Work

Over the past 10 years, a number of geometry libraries have been developed, each with its own notion of a geometry kernel. The C++ libraries PLAGEO and SPAGEO [17] provide kernels for 2- and 3-dimensional objects using floating point arithmetic, a class hierarchy, and a common base class. The C++ library LEDA [23] provides in its geometry part two kernels, one using exact rational arithmetic and the other floating point arithmetic. The Java library GEOMLIB [3] provides a kernel built in a hierarchical manner and designed around Java interfaces. None has addressed the questions of easily exchangeable and adaptable kernels.

Flexibility is one of the cornerstones of CGAL [10], the *Computational Geometry Algorithms Library*, which is being developed in a common project of several universities and research institutes in Europe and Israel. The recent overview [15] gives an extensive account of functionality, design, and implementation techniques in the library. Generic programming is one of the tools used to achieve this flexibility [7, 24, 25].

In the original design of the geometry kernel of CGAL [14], the geometric objects were each parameterized by a representation class, which was in turn parameterized by a number type. This design provided easy exchange of representation classes, was extensible, and provided limited adaptability of an existing representation class. However, the design did not allow the representation classes to be extended to also include geometric operations.

This extension was seen as desirable after the introduction of *geometric traits classes* into the library, which separate the combinatorial part of an algorithm or data structure from the underlying geometry. The term traits class was originally introduced by Myers [26]; we use it here to refer to a class that aggregates (geometric) types and operations. By supplying different traits classes, the same algorithm can be applied to different kinds of objects. Thus the use of traits classes brought about even more flexibility at a higher level in the library and, for example, allowed an easy means of comparison of different kernels in CGAL and LEDA using appropriate traits classes from CGAL [27].

As a kernel is generally considered to represent a basic set of building blocks for geometric computations, it is quite natural to assume that the kernel itself can be used as a traits class for many algorithms. This means that the concept of a kernel must include not only the representations for objects but also the operations on these objects, and for maximum flexibility both should be easily adaptable. Indeed, the fact that the existing CGAL kernel did not present its functionality in a way that was immediately accessible for the use in traits classes was one motivation for this work. Factoring out common requirements from the traits classes of different algorithms into the kernel is very helpful in maintaining uniform interfaces across a library and maximizing code reuse.

While the new design described here is even more flexible and more powerful than the old design, it maintains backwards compatibility. The kernel concept now includes easily exchangeable functors in addition to the geometric types; the ideas of traits classes and kernel representations have been unified. The implementation is accomplished by using a template programming idiom similar to the Barton-Nackman trick [5, 11] that uses a derived class as a template argument for a base class template. A similar idiom has been used in CGAL to solve cyclic template dependencies in the halfedge data structure and polyhedral surface design [21].

3 The Kernel Concept and Architecture

A geometry kernel consists of types used to represent geometric objects and operations on these types. Although from a C++ point of view both will be classes, we refer only to the former as (geometric) types whereas we call the latter (geometric) operations. Since different kernels will have different notions of what basic types and operations are required, we do not concern ourselves here with listing the particular objects and operations to be included in the kernel. Rather, we describe the kernel concept in terms of the interface it provides for each object and operation.

Depending on one's perspective, the expected interface to these types and operations will look somewhat different. From the point of view of an imperative-style programmer, it is natural that the types appear as stand-alone classes and the operations as global functions or member functions of these classes.

```
K::Point_2   p(0,1), q(1,-4);
K::Line_2    line(p, q);

if (less_xy_2(p, q)) { ... }
```

However, from the point of view of someone implementing algorithms in a generic way, it is most natural, indeed most useful, if types and operations are both provided by the kernel. This encapsulation allows both types and operations to be adapted and exchanged in the same manner.

```
K k;
K::Construct_line_2  c_line   = k.construct_line_2_object();
K::Less_xy_2         less_xy  = k.less_xy_2_object();
K::Point_2           p(0,1);
K::Point_2           q(1,-4);
K::Line_2            line = c_line(p, q);

if (less_xy(p, q)) { ... }
```

The concept of a kernel we introduce here includes both of these perspectives. That is, each operation is represented both as a type, an instance of which can be used like a function, and as a global function or a member function of one of the object classes. The techniques described in the following three sections allow both interfaces to coexist peacefully under one roof with a minimal maintenance overhead, and thus lead to a kernel that presents a good face to everyone.

Our kernel is constructed from three layers, illustrated in Figure 1. The bottom layer consists of basic numeric primitives such as the computation of matrix determinants and the construction of line equations from point coordinates. These numeric primitives are used in the geometric primitives that constitute the second layer of our structure. The top layer then assimilates the geometric primitives. The scope of our kernel concept is representation-independent affine geometry. Thus the concept includes, for example, the

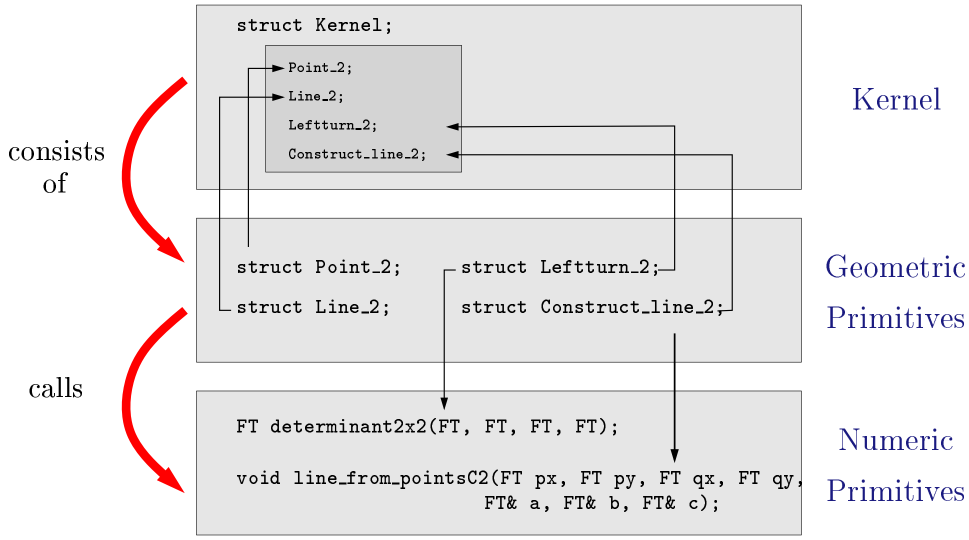

Figure 1: The kernel architecture.

construction of a point as the intersection of two lines but not its construction from x and y coordinates.

4 An Adaptable Kernel

We present our techniques using a simplified example kernel. Consider the types `Point_2` and `Line_2` representing two-dimensional points and lines, respectively, an operation `Construct_line_2` that constructs a `Line_2` from two `Point_2` arguments, and an operation `Less_xy_2` that compares two `Point_2` objects lexicographically. The kernel for these types and operations might then look as follows; the classes `MyPoint`, `MyLine`, `MyConstruct`, and `MyLess` are arbitrary and defined elsewhere.

```
struct Kernel {
  typedef MyPoint      Point_2;
  typedef MyLine       Line_2;
  typedef MyConstruct  Construct_line_2;
  typedef MyLess       Less_xy_2;
};
```

In general, one probably needs more operations and possibly more types in order to be able to do something useful, but for the sake of simplicity we will stay with these four items for the time being.

A first question might be: `Construct_line_2` has to construct a `Line_2` from two `Point_2`s; hence it has to know something about both types. How does it get to know them? Since we are talking about adaptability, just hard-wiring the names `MyPoint` and `MyLine` into `MyConstruct` is not what we would like to do.

A natural solution is to parameterize `MyConstruct` with the other classes, that is, with our kernel. As soon as a class knows the kernel it resides in, it also knows all related classes and operations. A straightforward way to implement this parameterization is to supply the kernel as a template argument to the geometric classes.

```
template < class K > struct MyPoint { ... };
template < class K > struct MyLine { ... };
template < class K > struct MyConstruct { ... };
template < class K > struct MyLess { ... };
```

Our kernel class from above has to be changed accordingly.

```
struct Kernel {
  typedef MyPoint< Kernel >      Point_2;
  typedef MyLine< Kernel >       Line_2;
  typedef MyConstruct< Kernel >  Construct_line_2;
  typedef MyLess< Kernel >       Less_xy_2;
};
```

At first, it might look a bit awkward; inserting a class into its own components seems to create cyclic references. Indeed, one has to be careful, as the following example demonstrates.

```
template < class T >
struct P {
  typedef typename T::A B;
};

struct K {
  typedef P< K >::B B; // *
  typedef int A;
};
```

A reference to `K::B` will lead to `P<K>::B` and further to `K::A`, but this type is not yet declared in line $*$. A reasonable C++ compiler will thus give up at that point. But there is no such problem with the `Kernel` class above; the class is considered to be declared as soon as the class name has been read (cf. [9] 9/2), hence it is fine to provide it as a template argument to other classes. The problem in class `K` came from the fact that `P<K>::B` refers back to `K` inside its own definition, to the still undefined type `K::A`.

Leaving these subtleties, let us come back to the main theme: adaptability. It should be easy to extend or adapt this kernel and indeed, all that needs to be done is to derive a new class from `Kernel` where new types can be added and existing ones can be exchanged.

```
struct New_kernel : public Kernel {
  typedef NewPoint< New_kernel >   Point_2;
  typedef MyLeftTurn< New_kernel > Left_turn_2;
};
```

Here `Point_2` is overwritten with a different type and the new operation `Left_turn_2` is defined. So let us start programming with the newly constructed kernel.

```
New_kernel::Point_2 p, q;
New_kernel::Construct_line_2 construct_line_2;
// initialize p, q and construct_line_2
New_kernel::Line_2 l = construct_line_2(p, q);
```

To our surprise and anger, the last line refuses to compile.

```
No instance of function "MyConstruct<Kernel>::operator()"
matches the argument list.
The argument types are: (New_kernel::Point_2, New_kernel::Point_2).
```

What has gone wrong? Apart from the fact that we did not show the implementation of `MyConstruct` yet and hence the reference to `operator()` is not clear, there is one thing that should catch our eyes: the compiler complains about `MyConstruct<Kernel>` whereas we would like to see `MyConstruct<New_kernel>`. On the other hand, this is not really surprising, since we did not change the type `Construct_line_2` in `New_kernel`, hence, it is the same as in `Kernel`, that is `MyConstruct<Kernel>`. `MyConstruct<>` uses the type `Kernel::Point_2` (= `MyPoint<>`) and cannot handle `New_kernel::Point_2` (= `NewPoint<>`) arguments properly; hence, the error message.

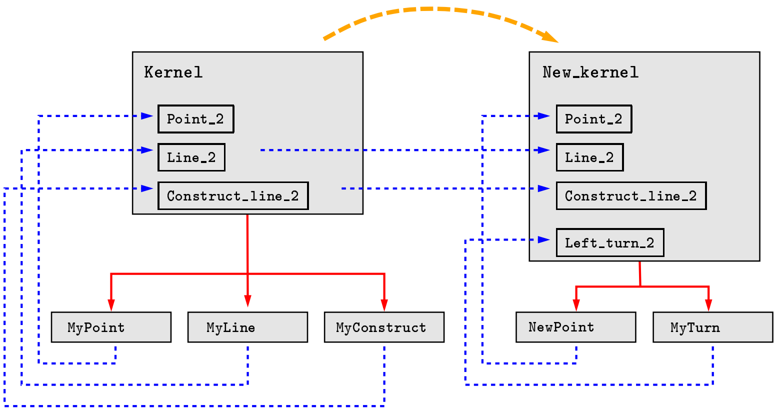

Figure 2: Instantiation problem. Boxes stand for classes, thick dashed arrows denote derivation, solid arrows show (template) parameters, and thin dotted arrows have to be read as “defines” (typedef or inheritance).

What can be done to tell `MyConstruct` that it should now consider itself part of `New_kernel`? An obvious solution would be to redefine `Construct_line_2` in `New_kernel` appropriately. This is fine in our example where it amounts to just one more typedef, but considering a real kernel with dozens of types and hundreds of operations, it would be really tedious to have to repeat all these definitions. Note that it may well be that these classes have to be redefined anyway, as the change of one class potentially affects all other classes that interact with that class. But often it is not necessary[1], and we do not want this redefinition as a general requirement.

[1]Consider replacing a class by another class providing the same interface or a superset of it, e.g., derived classes.

Fortunately, there is a way out. If Kernel is meant as a base for building custom kernel classes, it is not wise to instantiate MyPoint<>, MyLine<> and MyConstruct<> at that point with Kernel, as this might not be the kernel in which these classes finally end up. We rather would like to defer the instantiation, until it is clear what the actual kernel will be. This can be done by introducing a class Kernel_base that serves as an "instantiation-engine." Actual kernel classes like Kernel and New_kernel both derive from Kernel_base and finally start the instantiation by injecting themselves into the base class.

```
template < class K >
struct Kernel_base {
  typedef MyPoint< K >      Point_2;
  typedef MyLine< K >       Line_2;
  typedef MyConstruct< K >  Construct_line_2;
  typedef MyLess< K >       Less_xy_2;
};

struct Kernel : public Kernel_base< Kernel > {};
```

It seems somewhat strange to insert a class into its base class, that is into itself in some sense. But looking at it more closely quickly reveals that the construction is not much different from the previous one, except for giving the additional freedom to determine when MyPoint *etc.* are instantiated. It is still easy to create new kernels by derivation, now from Kernel_base. In order to be able to extend New_kernel in the same way as Kernel, we defer instantiation once again.

```
template < class K >
struct New_kernel_base : public Kernel_base< K > {
  typedef NewPoint< K >   Point_2;
  typedef MyLeftTurn< K > Left_turn_2;
};

struct New_kernel : public New_kernel_base< New_kernel > {};
```

Thus we achieve our easily extensible and adaptable kernel through the use of the kernel as a template parameter at two different levels. The geometric object classes in the kernel use the kernel as a template parameter so the distinct geometric objects have a way of discovering the types of the other objects and operations. Thus any change of a type or operation in the kernel is propagated through to the relevant object classes. And the kernel itself is derived from a base class that is templated by the kernel, which assures that the types and operations instantiated are the types in the derived class and not in the base class. Thus any modified types or operations live in the same kernel as the ones inherited from the base class and there is no problem in using the two together.

5 Functors

The question still remains how we provide the actual functions that are needed by the classes and functions that interact through the kernel. Consider again the example from the previous section:

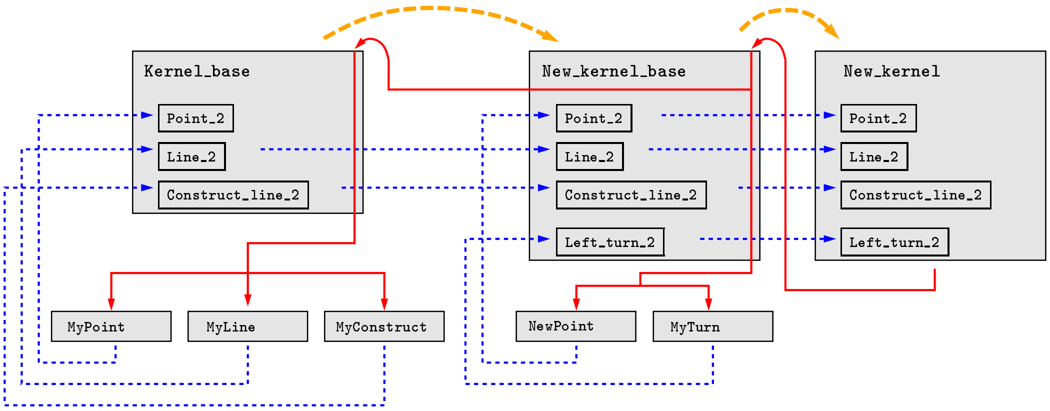

Figure 3: Deferring instantiation. Boxes stand for classes, thick dashed arrows denote derivation, solid arrows show (template) parameters, and thin dotted arrows have to be read as "defines" (typedef or inheritance).

```
New_kernel::Point_2 p, q;
New_kernel::Construct_line_2 construct_line_2;
// initialize p, q and construct_line_2
New_kernel::Line_2 l = construct_line_2(p, q);
```

What we are concerned with here is how the kernel provides the function `construct_line_2`. There are a number of ways such a function can be provided in a way that assures adaptability of the kernel. Adaptability is not the only concern, however. A real kernel will contain many constructions and predicates, most of them small, containing only a few lines of code. These functions will be called a huge number of times during the execution of an algorithm implemented on top of the kernel; they are to geometry what additions and multiplications are to arithmetics. Hence, efficiency is very important.

The classic C-style approach would be to use *pointers to functions* in the kernel.

```
struct Kernel {
  typedef Line_2 (*Construct_line_2)(Point_2 p, Point_2 q);
  Construct_line_2 construct_line_2;
};
```

Adaptability is provided by the ability to change the pointer (`construct_line_2` in our example). But the additional indirection when calling the function imposes a considerable performance penalty for small functions. We will demonstrate this behaviour below. *Virtual functions* are the Java-style means of achieving adaptability.

```
struct Kernel_base {
  virtual Line_2 construct_line_2(Point_2 p, Point_2 q);
};
```

As with pointers to functions, though, there is an additional indirection involved (lookup in the virtual function table); moreover, many compiler optimisations are not possible through virtual functions [31], as the actual types are not known at compile time. This overhead is considerable in our context [27].

So if virtual functions are too costly, how about making `construct_line_2` a plain member function of `Kernel_base`? The function can then be adapted by overwriting it in derived classes. Indeed, what we propose is just one step further, and involves moving from concrete function signatures in the programming language to a more abstract level. The solution is inspired by the standard `C++` library [9], where many algorithms are parameterized with so-called *function objects*, or *functors*. The crucial observation behind this abstraction is the following: it is not important whether something *is* a function, as long as it *behaves like* a function and thus can be used as a function. So what is the behaviour of a function? It is something you can call by using parentheses and passing arguments [19].

Obviously, any function is a functor. But objects of a class-type that define an appropriate `operator()` can be functors as well.

```
struct Construct_line_2 {
  Line_2 operator()(Point_2 p, Point_2 q) const
  { // build a line from points p and q; }
};
```

This way, any instance of `Construct_line_2` can be used as if it were a function.

```
Point_2 p, q;
Construct_line_2 construct_line_2;
Line_2 l = construct_line_2(p,q);
```

There are at least three advantages that make this abstraction worthwhile: efficiency, ability to maintain a state, and better type checking, all explained in more detail below. Although the first two advantages can be achieved by using plain member functions of the kernel class, there are a few reasons that functors are preferable.

- Geometric operations that are functors can be used together with algorithms from the standard library such as `sort`, `lower_bound`, *etc.*
- Functors are cleanly separated from each other and can maintain their states independently.
- Functors provide an almost uniform framework where both representations and operations are just types in the kernel class, and the mechanisms for adapting and exchanging them are the same.
- Functors provide a simpler calling syntax since it is independent of the kernel object, whereas member functions require the use of the kernel object in every call.

5.1 Efficiency of Functors

If the complete class definition for a functor is known at compile time, the `operator()` can be inlined. Handing the functor as a template argument to some function template or class template is like literally handing over a piece of code that can be inlined and optimized to the compiler's taste. Note again the contrast to the traditional function pointers and to virtual functions.

To support this claim, we have made a small test: sort 5000 `double` numbers with bubble-sort, beginning with the numbers in worst-case order. The first function compares numbers using the built-in `operator<`, the second is parameterized with a functor for the comparison, and the third with a function pointer. With compilers that optimize well, there is absolutely no difference in runtime between the generic functor and the "handcrafted" version, while the function pointer parameterization causes a considerable overhead; see Table 1.

System	Compiler	<	functor	pointer
Linux	`g++ -O3`	370	430	1150
Irix	`CC -Ofast`	760	760	1350
Solaris	`g++ -O3`	770	860	5360
Solaris	`CC -fast`	890	890	3300
Solaris	`KCC +K3`	740	740	2560

Table 1: Runtime in msec. to sort 5000 `double` numbers.

5.2 Functor with State

In addition to their potential for optimizations, functors also prove to be more flexible than plain functions; a functor of class-type has a state that can carry local data. While state could also be implemented using static variables in member or global functions, this would forbid working with more than one instance of the function, imposing a severe and hard-to-check restriction. And maintaining a single monolithic kernel object that aggregates the states of all its operations would be quite difficult to handle.

Let us assume that, for the purposes of benchmarking, we want to count the number of comparisons done by a program using our functor `MyLess` in the sample kernel `Kernel` above.

```
template <class K>
struct MyLess {
  typedef typename K::Point_2   Point_2;

  int* count;
  MyLess(int* counter) : count(counter) {}
  bool operator()(Point_2 p, Point_2 q) const {
    ++(*count);
    return p.x < q.x || p.x == q.x && p.y < q.y;
  }
};
```

Each call to this functor increases the externally referenced counter by one. Other, more serious, examples of functors using a state are the adaptors `binder1st` and `binder2nd` in the STL. They use a local variable to store the value to which one of a functor's arguments gets bound. Also the projection traits described in Section 8.3 needs a state to store the projection direction.

Allowing local data for a functor adds a slight complication to the kernel. Clearly, a generic algorithm has to be oblivious to whether a functor carries local state or not. Hence, the algorithm cannot instantiate the functor itself. As the example above illustrates, a function with local state may require the use of a non-default constructor while one without a local state does not. But we can assume that the kernel knows how to create functors. So we add access member functions to the kernel that allow a generic algorithm to obtain an object for a functor. Here is the revised kernel base class for the example from the previous section. The access member functions are simply inherited by all derived kernels and kernel base classes.

```
template < class K >
struct Kernel_base {
  typedef MyPoint< K >      Point_2;
  typedef MyLine< K >       Line_2;
  typedef MyConstruct< K >  Construct_line_2;
  typedef MyLess< K >       Less_xy_2;
  Construct_line_2          construct_line_2_object();
  Less_xy_2                 less_xy_2_object();
};
```

The actual implementations of `construct_line_2_object` and `less_xy_2_object` depend on `MyConstruct` and `MyLess`, respectively, and might be as simple as the default constructors.

5.3 Better Type Matching

The type of a function is defined by its signature, while the types of general functors can be as different as one likes. This is an advantage in template argument matching, as there is more freedom in expressing the set of matching types. Consider, for example, a data structure `D<>` that is parameterized with a predicate, and imagine a class `P<>` of predicates for which you would like to share the implementation of `D< P<> >`. By defining a specialization

```
template < class T > struct D< P< T > > { ... };
```

this code sharing is easily accomplished, while doing so on the level of function signatures is not possible in a straightforward manner.

6 An Imperative Interface

Someone used to imperative-style programming might expect a kernel interface based on member functions and global functions operating on the geometric classes rather than having to deal with functors and kernel objects. Due to the flexibility in our design, we can easily provide such an interface on top of the kernel with little overhead. However, some care has to be taken, such that the genericity is not lost in this step.

Consider again the operation for determining if one point is lexicographically smaller than another. We have provided this operation through our kernel with the type `Less_xy_2` and the member function `less_xy_2_object()`, which creates an instance of the functor. It is also quite natural to provide this operation as a global function in our kernel interface. In order to handle correctly functors with state, a kernel object has to be a parameter of such a function. A default argument can be used such that the kernel object does not have to be provided where the default kernel suffices.

```
template < class K >
bool less_xy_2(typename K::Point_2 p, typename K::Point_2 q, K k = K())
{ return k.less_xy_2_object()(p, q); }
```

However, if the kernel parameter `k` is omitted, the type `K` cannot be deduced from the actual parameters of the function call (cf. [9] 14.8.2.1/4). Hence, the template parameter has to be specified explicitly in this case.

```
Kernel::Point_2 p, q;
if (less_xy_2<Kernel>(p, q)) { ... }
```

While such functions allow one to write completely generic code, one might still object to the spurious-looking `<Kernel>` parameter in the global function call. It would be preferable to be able to avoid this parameter in some cases, e.g., where only one specific kernel is ever used. The solution is to overload the function for parameters from this specific kernel.

```
bool less_xy_2(Point_2< Default_kernel_1 > p,
               Point_2< Default_kernel_1 > q)
{ return less_xy_2<Default_kernel_1>(p, q); }
```

Note that these specialized functions can be templated again, e.g., by a number type, as long as they are not templated with the kernel class.[2] Then both the specialized function and the function with the kernel template parameter can peacefully coexist, and also both ways of calling them can be used simultaneously.

One might also want to add some functionality to the geometric types. For example, if the kernel supports the construction of a line from two points, it is natural that the class `MyLine` has a constructor that takes two point arguments.

```
template < class K >
struct MyLine {
  MyLine(typename K::Point_2 p, typename K::Point_2 q)
  { ... use e.g. K::Construct_line_2 ... }
};
```

Again it is important that `MyLine` does not make assumptions about the point type, but uses the operations provided by `K` only. This way, the geometric types remain nicely separated, as their – sometimes close – relationships are encapsulated into appropriate operations.

[2]If they were, the call `less_xy_2<Default_kernel_1>(...)` would be ambiguous.

7 A Function Toolbox

Our kernel concept nicely separates the representation of geometric objects from the operations on these objects. But when implementing a specific operation such as `Left_turn_2`, the representation of the corresponding point type `Point_2` will inevitably come into play; in the end, the predicate is evaluated using arithmetic operations on some number type. The nontrivial[3] algebraic computations needed in predicates and constructions are encapsulated in the bottom layer of our kernel architecture (Figure 1), the *number-type-based function toolbox*, which we describe in this section.

A *number type* refers to a numerical type that we use to store coordinates and to calculate results. Given that the coordinates we start with are rational numbers, it suffices to compute within the domain of rational numbers. For certain operations we will go beyond rational arithmetic and require roots. However, since the majority of our kernel requires only rational arithmetic we focus on this aspect here. Depending on the calculations required for certain operations, we distinguish between different concepts of number types that are taken from algebra. A *ring* supports addition, subtraction and multiplication. A *Euclidean ring* supports the three ring operations and an integral division with remainder, which allows the calculation of greatest common divisors used, *e.g.*, to cancel common factors in fractions. In contrast, a *field* type supports exact division instead of integral division.

Many of the operations in our kernel boil down to determinant evaluations, *e.g.*, sidedness tests, in-circle tests, or segment intersection. For example, the left-turn predicate is evaluated by computing the sign of the determinant of a 2×2 matrix built from differences of the points' coordinates. Since the evaluation of such a determinant is needed in several other predicates as well, it makes sense to factor out this step into a separate function, which is parameterized by a number type (here `FT` for field type) to maintain flexibility even at this level of the kernel:

```
template < class FT >
FT determinant2x2(FT a00, FT a01, FT a10, FT a11)
{ return a00 * a11 - a10 * a01; }
```

The function can now be shared by all predicates and constructions that need to evaluate a 2×2 determinant. This code reuse is desirable not only because it reduces maintenance overhead but also from a robustness point of view, as it isolates potential problems in a small number of places. And this also enhances the adaptability and extensibility of our kernel. These basic numerical operations are equally as accessible to anyone providing additional or customized operations on top of our kernel in the future.

8 Adaptable Algorithms

In the previous sections, we have illustrated the techniques used to realize a kernel concept that includes functors as well as types in a way that makes both easily adaptable. Here we show how such a kernel can be put to good use in the implementation and adaptation of an algorithm.

[3] beyond a single addition or comparison

In CGAL, the geometric requirements of an algorithm are collected in a geometric traits class that is a template parameter for the algorithm. With the addition of functors to the kernel concept, it is now possible simply to supply a kernel as the argument for the geometric traits class of an algorithm. And it is also now quite easy to replace a type or predicate provided with one of the kernels in CGAL with another, customized type or predicate and then use the adapted kernel as the traits class argument. We illustrate these points below.

In general, the requirements of many geometric traits classes are only a subset of the requirements of a kernel. Other geometric traits classes might have requirements that are not part of the kernel concept. They can be implemented as extensions on top, having easy access to the part of their functionality that is provided by the kernel.

8.1 Kernel as a Traits Class

Let us consider as a simple example Andrew's variant of Graham's scan [1, 12] for computing the convex hull of a set of points in two dimensions. This algorithm requires only a point type, the lexicographical comparison of points, and a left-turn predicate from its traits class. Thus, the kernel `New_kernel` from Section 4 suffices for this algorithm.

The function that implements this algorithm takes a range of random-access iterators providing the input sequence of points and a bidirectional iterator for the resulting sequence of hull points. The last argument is the traits class, that is, our kernel. For a thorough description of the standard iterator concepts refer to the book of Austern [2] or the online reference of SGI's STL [29]. Informally speaking, one can think of random-access iterators as pointers to an array, while bidirectional iterators can be regarded as pointers to a doubly-linked list.

Let us flesh out the example of the convex hull algorithm and see how it could be implemented[4]. The algorithm computes the convex hull and copies all points on the boundary of the convex hull (not only its corners) in counterclockwise order to the iterator `result`. It runs in $\mathcal{O}(n \log n)$ time, for a set of n input points, using linear space and can produce up to $2\,n-2$ output points in the degenerate case that all points are collinear.

```
template < class RandomAccessIterator,
           class BidirectionalIterator,
           class Traits >
BidirectionalIterator
ch_graham_andrew_scan(RandomAccessIterator first,
                      RandomAccessIterator beyond,
                      BidirectionalIterator result,
                      const Traits& traits)
{
  typename Traits::Left_turn_2 left_turn_2 = traits.left_turn_2_object();

  // lexicographical sorting + remove duplicates
  std::sort(first, beyond, traits.less_xy_2_object());
  beyond = std::unique(first, beyond, std::not2(traits.less_xy_2_object()));
```

[4]The implementation provided in CGAL is somewhat different.

```
  // lower convex hull (left to right)
  result = copy_if_triple_2(first, beyond, result, left_turn_2);

  // upper convex hull (right to left)
  typedef std::reverse_iterator< RandomAccessIterator > Rev;
  result = copy_if_triple_2(Rev(beyond), Rev(first), --result, left_turn_2);
  return --result;
}
```

Note that the implementation is very simple and concise due to the use of algorithms and data structures from the standard C++ library. It also uses the following function that, although non-standard, is heavily inspired by standard algorithms such as `std::remove_if` and `std::unique`.

```
template < class ForwardIterator, class RandomAccessIterator, class Predicate >
RandomAccessIterator
copy_if_triple_2(ForwardIterator first,
                 ForwardIterator beyond,
                 RandomAccessIterator result,
                 Predicate pred)
// copy a subrange of [f, b) to r, s.t. for any 3 consecutive elmts p
// is true. The subrange is obtained by successively removing the 2nd
// element from the 1st triple in [f, b) not satisfying p.
{
  *result = *first, ++result, ++first;
  RandomAccessIterator o = result;
  *result = *first, ++result, ++first;
  for (; first != beyond; ++result, ++first) {
    while (result != o && pred(*first, result[-1], result[-2]))
      --result;
    *result = *first;
  }
  return result;
}
```

Calling the algorithm with a kernel is straightforward. We can simplify the call further and hide the kernel parameter with a default argument. For the default we choose the kernel used for the points of the input sequence. We obtain the point type using `std::iterator_traits` and use the same technique in `Kernel_traits`, to deduce the kernel of a geometric object. Note that for this mechanism the kernel has to be default constructible.

```
template < class BidirectionalIterator, class OutputIterator >
OutputIterator
ch_graham_andrew_scan(BidirectionalIterator first,
                      BidirectionalIterator beyond,
                      OutputIterator result)
{
```

```
  typedef typename std::iterator_traits< BidirectionalIterator >::value_type P;
  typedef typename Kernel_traits< P >::Kernel Kernel;
  return ch_graham_andrew_scan(first, beyond, result, Kernel());
}
```

The class `Kernel_traits` is modelled after `std::iterator_traits`. The default implementation could be as follows.

```
template < class T >
struct Kernel_traits {
  typedef typename T::Kernel Kernel;
};
```

We assume a convention that points provide a local type `Kernel` for their kernel. For geometric classes that do not provide this type one has to define appropriate specializations of `Kernel_traits`.

8.2 Adapting a Predicate

Assume we use the convex hull function from above with a kernel that represents points by their Cartesian coordinates of type `double`[5]. The left-turn predicate amounts to evaluating the sign of a 2×2-determinant; if this is done in the straightforward way by calculations with `doubles`, the result is not guaranteed to be correct due to roundoff errors caused by the limited precision. It cannot be stressed enough, that this is not just a question of some minor errors in the output, *i.e.*, some points close to the boundary of the convex hull being classified wrongly; the whole combinatorics can break down, causing the algorithm to output garbage or even to loop endlessly.

While there is an easy way out, that is, using an exact number type [8, 20] instead of `double`, this often has to be paid for with a considerable loss in performance. An in-between solution is to do the calculations on the fast floating point type and calculate an error-bound from which one can deduce whether the result is correct, *i.e.*, the sign of the expression is known. Exact arithmetic is only used in those cases where the floating point calculation is not known to give the correct results, and the hope is that this happens seldom. The described technique is called *floating point filtering* [6, 16, 28], and depending on how the error bound is computed, one refers to the filters as static, semi-static or dynamic.

We will now describe how to adapt the kernel to use a statically filtered left-turn predicate, using the types `double` and some arbitrary-precision number type, which we call `exact`. Assume, we know that the coordinates of the input points are `double` values from $(-1, 1)$. It can be shown (cf. [28]) that in this case the correct sign can be determined from the `double` calculation, if the absolute value of the result exceeds

$$3 \cdot (2^{-50} + 2^{-102}) < \texttt{2.6645352591003765e-15}\ .$$

```
template < class K >
struct Static_filter_left_turn_2 {
```

[5] A double precision floating point number type as defined in IEEE 754 [18].

```
  typedef typename K::Point_2 Point_2;
  bool operator()(Point_2 p, Point_2 q, Point_2 r) const {
    // compute approximation
    double a = determinant2x2(q.x - p.x, q.y - p.y,
                              r.x - p.x, r.y - p.y);

    // test for error bound:
    const double epsilon = 2.6645352591003765e-15;
    if (a < -epsilon) return false;
    if (a > epsilon)  return true;

    // else compute exactly ...
    exact epx = p.x, epy = p.y;
    exact eqx = q.x, eqy = q.y;
    exact erx = r.x, ery = r.y;

    return determinant2x2(eqx - epx, eqy - epy,
                          erx - epx, ery - epy) > exact(0);
  }
}
```

Inserting this into our kernel is straightforward.

```
struct Filtered_kernel : public Kernel_base< Filtered_kernel > {
  typedef Static_filter_left_turn_2< Filtered_kernel > Left_turn_2;
  Left_turn_2 left_turn_2_object() const { return Left_turn_2(); }
};
```

And supplying this adapted kernel to the convex hull function will guarantee that the correct result is produced.

The example given here is specific for two particular number types (`double` and `exact`) and for a particular range of values for the coordinates. Thus, though useful, our adapted predicate is not applicable in all cases. In Section 9, we describe a model for a kernel provided in CGAL that is parameterized by two number types and automatically filters all predicates of a given, unfiltered kernel, although using a different method than the one illustrated above.

8.3 Projection Traits

As mentioned in Section 5, one benefit of using functors in the traits class and kernel class is the possible association of a state with the functor. This flexibility can be used, for example, to apply a two-dimensional algorithm to a set of coplanar points in three dimensions. Consider the problem of triangulating a set of points on a polyhedral surface. Each face of the surface can be triangulated separately using a two-dimensional triangulation algorithm and a kernel can be written whose two-dimensional part realizes the projection of the points onto the plane of the face in all functors while actually using the original three-dimensional data. The predicates must therefore know about the plane in which they are operating and this is maintained by the functors in a state variable.

9 Kernel Models

The techniques described in the previous sections have been used to realize several models for the geometry kernel concept described in Section 3. In fact, we use class templates to create a whole *family* of models at once. The template parameter is usually the number type used for coordinates and arithmetic (Section 7). We categorize our kernel families according to *coordinate representation*, *object reference and construction*, and *level of runtime optimization*. Furthermore, we have actually two kernel concepts in CGAL: a lower-dimensional kernel concept for the fixed dimensions 2 and 3, and a higher-dimensional kernel concept for arbitrary dimension d. For more details beyond what can be presented here, the reader is referred to the CGAL reference manuals [10].

9.1 Coordinate Representation

We distinguish two coordinate representations: Cartesian and homogeneous. The Cartesian representation is a class template `Cartesian<FT>` with the template parameter `FT` indicating the requirements for a *field type*. The homogeneous representation is a class template `Homogeneous<RT>` with the template parameter `RT` indicating the requirements for a *ring type*. Homogeneous representation allows many operations to factor out divisions into a common denominator, thus avoiding divisions in the computation, which can sometimes improve efficiency and robustness greatly. The Cartesian representation, however, avoids the extra time and space overhead required to maintain the homogenizing coordinate and thus can also be more efficient for certain applications.

9.2 Memory Allocation and Construction

An additional facet of optimization is the memory layout of the geometric objects. The standard technique of *smart pointers* can be used to speed up copy constructions and assignments of objects with a reference-counted handle-representation scheme. Runtime experiments show that this scheme pays off for objects whose size is larger than a certain threshold (around 4 words depending on the machine architecture). To allow for an optimal choice, CGAL offers for each representation a simple and a smart-pointer based version. In the Cartesian case, these models are called `Simple_cartesian<FT>` and `Cartesian<FT>`.

9.3 Filtered Models

The established approach for robust geometric algorithms following the exact computation paradigm [32] requires the exact evaluation of geometric predicates, *i.e.*, decisions derived from geometric computations have to be correct. While this can be achieved straightforwardly by relying on an exact number type, this is not the most efficient approach, and the idea of so-called *filters* has been developed to speed up the exact evaluation of predicates [6, 16, 28]. See also the example in Section 8.

The basic idea is to use a filtering step before the costly computation with an exact number type. The filter step evaluates quickly and approximately the result of the predicate, but is also able to decide if the answer it gives is certified to be true or if there is a risk for a false answer, in which case the exact number type is used to find the correct answer.

CGAL implements such a filtering technique using interval arithmetic, via the number type `Interval_nt` [6]. This number type stores an interval of two `double` values that changes to reflect the round-off errors that occur during floating point computations. The comparison operators on this number type have the property that they throw a C++ exception in case that the two intervals to be compared overlap. When this occurs, it means that the filter cannot certify the exactness of the result using its approximate computation. Then we have to find a different method to evaluate exactly the predicate, by using an exact, but slower, number type. As this failure is supposed to happen rarely on average, the overall performance of using the filtering is about the same as the evaluation of the predicate over the intervals, which is pretty fast.

CGAL provides an adaptor `Filter_predicate<>`, which makes it easy to use the filter technique for a given predicate, and also a full kernel `Filtered_kernel<>` with all predicates filtered using the scheme presented above.

Here is an example of an application to the orientation predicate. The functor `Cartesian<FT>::Orientation_2` is templated by a field type. This allows us to build the filtered version of the orientation predicate easily, provided we have an exact number type like `leda_real`. We simply define one version of the predicate with the interval number type as the field type and one with the exact number type and use both of these to define our filtered predicate.

```
typedef Cartesian< Interval_nt >::Orientation_2  Approx;
typedef Cartesian< leda_real >::Orientation_2    Exact;
typedef Filter_predicate< Approx, Exact >        Filter;
typedef Cartesian< double >::Point_2             Point;

{
  Point p(1.0, 2.0), q(2.0, 3.0), r(3.0, 4.0);
  return Filter()(p, q, r);
}
```

`Filter_predicate<>` has default template parameters specifying how to convert a `Point` to a `Cartesian<Interval_nt>::Point_2` in order to call the approximate version, and similarly in order to convert a `Point_2` to a `Cartesian<leda_real>::Point_2` for the eventual exact computation.

9.4 Higher-dimensional Kernel

The higher-dimensional kernel defines a concept with the same type and functor technology, but is well separated from the lower-dimensional kernel concepts. Higher-dimensional affine geometry is strongly connected to its mathematical foundation in linear algebra and analytical geometry. Therefore, a central task is the implementation and integration of a generic linear algebra module. Since the dimension is now a parameter of the interface and since the solution of linear systems can be done in different ways [13, 4, 22], a linear algebra concept is part of the interface of the higher dimensional kernel models `Cartesian_d<FT,LA>` and `Homogeneous_d<RT,LA>`. The linear algebra concept provides a standard interface to matrix and vector types and the solution of linear systems of equations.

10 Conclusions

Many of the ideas presented here have already been realized in CGAL; parts of them still need to be implemented. Although standard compliance is still a big issue for C++ compilers, more and more compilers are able to accept template code such as ours.

We would like to remind the reader that in this paper we have lifted the curtain to how to implement a library, which is considerably more involved than using a library. A user of our design can be gradually introduced to the default use of one kernel, then exchanging one kernel with another kernel in an algorithm, exchanging individual pieces in a kernel, and finally – for experts – writing a new kernel. Only creators of a new library need to know all inner workings of a design, but we believe also interested users will benefit from studying the design.

Acknowledgments

This work has been supported by ESPRIT LTR projects No. 21957 (CGAL) and No. 28155 (GALIA). The second author also acknowledges support from the Swiss Federal Office for Education and Science (CGAL and GALIA).

Many more people have been involved in the CGAL project, and contributed in one or the other way to the discussion that finally lead to the design presented here. We thank especially Hervé Brönnimann, Bernd Gärtner, Stefan Schirra, Wieger Wesselink, and Mariette Yvinec for their valuable input.

References

[1] ANDREW, A. M. Another efficient algorithm for convex hulls in two dimensions. *Inform. Process. Lett. 9*, 5 (1979), 216–219.

[2] AUSTERN, M. H. *Generic Programming and the STL*. Addison-Wesley, 1998.

[3] BAKER, J. E., TAMASSIA, R., AND VISMARA, L. GeomLib: Algorithm engineering for a geometric computing library, 1997. (Preliminary report).

[4] BARREIS, E. Computational solutions of matrix problems over an integral domain. *J. Inst. Maths Applications 10* (1972), 68–104.

[5] BARTON, J. J., AND NACKMAN, L. R. *Scientific and Engineering C++*. Addison-Wesley, Reading, MA, 1997.

[6] BRÖNNIMANN, H., BURNIKEL, C., AND PION, S. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.* (1998), pp. 165–174.

[7] Brönnimann, H., Kettner, L., Schirra, S., and Veltkamp, R. Applications of the generic programming paradigm in the design of CGAL. In *Generic Programming—Proceedings of a Dagstuhl Seminar* (2000), M. Jazayeri, R. Loos, and D. Musser, Eds., LNCS 1766, Springer-Verlag.

[8] Burnikel, C., Mehlhorn, K., and Schirra, S. The LEDA class real number. Technical Report MPI-I-96-1-001, Max-Planck Institut Inform., Saarbrücken, Germany, Jan. 1996.

[9] International standard ISO/IEC 14882: Programming languages – C++. American National Standards Institute, 11 West 42nd Street, New York 10036, 1998.

[10] CGAL, the Computational Geometry Algorithms Library. `http://www.cgal.org/`.

[11] Coplien, J. O. Curiously recurring template patterns. *C++ Report* (Feb. 1995), 24–27.

[12] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[13] Edmonds, J. Systems of distinct representatives and linear algebra. *Journal of Research of the National Bureau of Standards 71(B)* (1967), 241–245.

[14] Fabri, A., Giezeman, G.-J., Kettner, L., Schirra, S., and Schönherr, S. The CGAL kernel: A basis for geometric computation. In *Proc. 1st ACM Workshop on Appl. Comput. Geom.* (1996), M. C. Lin and D. Manocha, Eds., vol. 1148 of *Lecture Notes Comput. Sci.*, Springer-Verlag, pp. 191–202.

[15] Fabri, A., Giezeman, G.-J., Kettner, L., Schirra, S., and Schönherr, S. On the design of CGAL, the computational geometry algorithms library. *Software – Practice and Experience 30* (2000), 1167–1202.

[16] Fortune, S., and Van Wyk, C. J. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph. 15*, 3 (July 1996), 223–248.

[17] Giezeman, G.-J. *PlaGeo, a library for planar geometry, and SpaGeo, a library for spatial geometry*. Utrecht University, 1994.

[18] *IEEE Standard for binary floating point arithmetic, ANSI/IEEE Std* 754−1985. New York, NY, 1985. Reprinted in SIGPLAN Notices, 22(2):9–25, 1987.

[19] Josuttis, N. M. *The C++ Standard Library, A Tutorial and Reference*. Addison-Wesley, 1999.

[20] Karamcheti, V., Li, C., Pechtchanski, I., and Yap, C. *The CORE Library Project*, 1.2 ed., 1999. http://www.cs.nyu.edu/exact/core/.

[21] KETTNER, L. Using generic programming for designing a data structure for polyhedral surfaces. *Comput. Geom. Theory Appl. 13* (1999), 65–90.

[22] MCCLELLAN, MICHAEL T. The Exact Solution of Systems of Linear Equations with Polynomial Coefficients. *JACM 20*, 4 (October 1973), 563–588.

[23] MEHLHORN, K., AND NÄHER, S. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, UK, 1999.

[24] MUSSER, D. R., AND STEPANOV, A. A. Generic programming. In *1st Intl. Joint Conf. of ISSAC-88 and AAEC-6* (1989), Springer LNCS 358, pp. 13–25.

[25] MUSSER, D. R., AND STEPANOV, A. A. Algorithm-oriented generic libraries. *Software – Practice and Experience 24*, 7 (July 1994), 623–642.

[26] MYERS, N. C. Traits: A new and useful template technique. *C++ Report* (June 1995). `http://www.cantrip.org/traits.html`.

[27] SCHIRRA, S. A case study on the cost of geometric computing. In *Proc. Workshop on Algorithm Engineering and Experimentation* (1999), vol. 1619 of *Lecture Notes Comput. Sci.*, Springer-Verlag, pp. 156–176.

[28] SHEWCHUK, J. R. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete Comput. Geom. 18*, 3 (1997), 305–363.

[29] Standard Template Library programmer's guide. `http://www.sgi.com/tech/stl/`.

[30] STROUSTRUP, B. *The C++ Programming Language, 3rd Edition.* Addison-Wesley, 1997.

[31] VELDHUIZEN, T. Techniques for scientific C++. Technical Report 542, Department of Computer Science, Indiana University, 2000. `http://www.extreme.indiana.edu/~tveldhui/papers/techniques/`.

[32] YAP, C. K., AND DUBÉ, T. The exact computation paradigm. In *Computing in Euclidean Geometry*, D.-Z. Du and F. K. Hwang, Eds., 2nd ed., vol. 4 of *Lecture Notes Series on Computing.* World Scientific, Singapore, 1995, pp. 452–492.

A A Simple Example Kernel

```
// ---------------------------------------------------------
// bottom layer: number type based function toolbox
//

template < class FT >
FT determinant2x2(FT a00, FT a01, FT a10, FT a11)
{ return a00 * a11 - a10 * a01; }

template < class FT >
```

```
void line_from_pointsC2(FT px, FT py, FT qx, FT qy,
                        FT& a, FT& b, FT& c)
{
  a = py - qy;
  b = qx - px;
  c = -px * a - py * b;
}

// ---------------------------------------------------------
// mid layer: representations, predicates and constructions
//

template < class K_ >
struct Point_2 {
  typedef K_ K;
  typedef typename K::FT FT;
  Point_2() {}
  Point_2(FT x_, FT y_) : x(x_), y(y_) {}
  FT x, y;
};

template < class K_ >
struct Line_2 {
  typedef K_ K;
  typedef typename K::Point_2 Point_2;
  Line_2() {}
  Line_2(Point_2 p, Point_2 q)
  { *this = K::Construct_line_2(p, q); }
  typename K::FT a, b, c;
};

template < class K_ >
struct Segment_2 {
  typedef K_ K;
  typename K::Point_2 s, e;
};

template < class K_ >
struct Less_xy_2 {
  typedef typename K_::Point_2 Point_2;
  bool operator()(Point_2 p, Point_2 q) const
  { return p.x < q.x || p.x == q.x && p.y < q.y; }
};

template < class K_ >
struct Left_turn_2 {
  typedef typename K_::Point_2 Point_2;
  bool operator()(Point_2 p, Point_2 q, Point_2 r) const
```

```
  {
    return determinant2x2(q.x - p.x, q.y - p.y,
                          r.x - p.x, r.y - p.y) > 0;
  }
};

template < class K_ >
struct Construct_line_2 {
  typedef typename K_::Point_2 Point_2;
  typedef typename K_::Line_2  Line_2;
  Line_2 operator()(Point_2 p, Point_2 q) const {
    Line_2 l;
    line_from_pointsC2(p.x, p.y, q.x, q.y, l.a, l.b, l.c);
    return l;
  }
};

// ---------------------------------------------------------
// top layer: geometric kernel
//

template < class K_, class FT_ >
struct Kernel_base {
  typedef K_                      K;
  typedef FT_                     FT;
  typedef Point_2< K >            Point_2;
  typedef Line_2< K >             Line_2;
  typedef Segment_2< K >          Segment_2;
  typedef Less_xy_2< K >          Less_xy_2;
  typedef Left_turn_2< K >        Left_turn_2;
  typedef Construct_line_2< K > Construct_line_2;

  Less_xy_2        less_xy_2_object() const
  { return Less_xy_2(); }
  Left_turn_2      left_turn_2_object() const
  { return Left_turn_2(); }
  Construct_line_2 construct_line_2_object() const
  { return Construct_line_2(); }
};

template < class FT_ >
struct Kernel : public Kernel_base< Kernel< FT_ >, FT_ >
{};

// ---------------------------------------------------------
// convenience layer: global functions
//
```

```
template < class K > inline
bool
less_xy_2(typename K::Point_2 p, typename K::Point_2 q, K k = K())
{ return k.less_xy_2_object()(p, q); }

template < class K > inline
bool
left_turn_2(typename K::Point_2 p,
            typename K::Point_2 q,
            typename K::Point_2 r,
            K k = K())
{ return k.left_turn_2_object()(p, q, r); }

// ---------------------------------------------------------
// even more convenience: specializations for Kernel
//

template < class FT > inline
bool
left_turn_2(Point_2< Kernel< FT > > p,
            Point_2< Kernel< FT > > q,
            Point_2< Kernel< FT > > r)
{ return left_turn_2(p, q, r, Kernel< FT >()); }

template < class FT > inline
bool
less_xy_2(Point_2< Kernel< FT > > p, Point_2< Kernel< FT > > q)
{ return less_xy_2(p, q, Kernel< FT >()); }
```

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2001-4-005	H.P.A. Lensch, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2001-4-004	S.W. Choi, H. Seidel	Linear One-sided Stability of MAT for Weakly Injective Domain
MPI-I-2001-4-003	K. Daubert, W. Heidrich, J. Kautz, J. Dischler, H. Seidel	Efficient Light Transport Using Precomputed Visibility
MPI-I-2001-4-002	H.P.A. Lensch, J. Kautz, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing, Transmission, and Interactive Display of High Quality 3D Models on the Web
MPI-I-2001-4-001	H.P.A. Lensch, J. Kautz, M. Goesele, W. Heidrich, H. Seidel	Image-Based Reconstruction of Spatially Varying Materials
MPI-I-2001-2-005	V. Sofronie-Stokkermans	Resolution-based decision procedures for the universal theory of some classes of distributive lattices with operators
MPI-I-2001-2-004	H. de Nivelle	Translation of Resolution Proofs into Higher Order Natural Deduction using Type Theory
MPI-I-2001-2-003	S. Vorobyov	Experiments with Iterative Improvement Algorithms on Completely Unimodel Hypercubes
MPI-I-2001-2-002	P. Maier	A Set-Theoretic Framework for Assume-Guarantee Reasoning
MPI-I-2001-2-001	U. Waldmann	Superposition and Chaining for Totally Ordered Divisible Abelian Groups
MPI-I-2001-1-004	S. Hert, M. Hoffmann, L. Kettner, S. Pion, M. Seel	An Adaptable and Extensible Geometry Kernel
MPI-I-2001-1-003	M. Seel	Implementation of Planar Nef Polyhedra
MPI-I-2001-1-002	U. Meyer	Directed Single-Source Shortest-Paths in Linear Average-Case Time
MPI-I-2001-1-001	P. Krysta	Approximating Minimum Size 1,2-Connected Networks
MPI-I-2000-4-003	S.W. Choi, H. Seidel	Hyperbolic Hausdorff Distance for Medial Axis Transform
MPI-I-2000-4-002	L.P. Kobbelt, S. Bischoff, K. Kähler, R. Schneider, M. Botsch, C. Rössl, J. Vorsatz	Geometric Modeling Based on Polygonal Meshes
MPI-I-2000-4-001	J. Kautz, W. Heidrich, K. Daubert	Bump Map Shadows for OpenGL Rendering
MPI-I-2000-2-001	F. Eisenbrand	Short Vectors of Planar Lattices Via Continued Fractions
MPI-I-2000-1-005	M. Seel, K. Mehlhorn	Infimaximal Frames A Technique for Making Lines Look Like Segments

MPI-I-2000-1-004	K. Mehlhorn, S. Schirra	Generalized and improved constructive separation bound for real algebraic expressions
MPI-I-2000-1-003	P. Fatourou	Low-Contention Depth-First Scheduling of Parallel Computations with Synchronization Variables
MPI-I-2000-1-002	R. Beier, J. Sibeyn	A Powerful Heuristic for Telephone Gossiping
MPI-I-2000-1-001	E. Althaus, O. Kohlbacher, H. Lenhof, P. Müller	A branch and cut algorithm for the optimal solution of the side-chain placement problem
MPI-I-1999-4-001	J. Haber, H. Seidel	A Framework for Evaluating the Quality of Lossy Image Compression
MPI-I-1999-3-005	T.A. Henzinger, J. Raskin, P. Schobbens	Axioms for Real-Time Logics
MPI-I-1999-3-004	J. Raskin, P. Schobbens	Proving a conjecture of Andreka on temporal logic
MPI-I-1999-3-003	T.A. Henzinger, J. Raskin, P. Schobbens	Fully Decidable Logics, Automata and Classical Theories for Defining Regular Real-Time Languages
MPI-I-1999-3-002	J. Raskin, P. Schobbens	The Logic of Event Clocks
MPI-I-1999-3-001	S. Vorobyov	New Lower Bounds for the Expressiveness and the Higher-Order Matching Problem in the Simply Typed Lambda Calculus
MPI-I-1999-2-008	A. Bockmayr, F. Eisenbrand	Cutting Planes and the Elementary Closure in Fixed Dimension
MPI-I-1999-2-007	G. Delzanno, J. Raskin	Symbolic Representation of Upward-closed Sets
MPI-I-1999-2-006	A. Nonnengart	A Deductive Model Checking Approach for Hybrid Systems
MPI-I-1999-2-005	J. Wu	Symmetries in Logic Programs
MPI-I-1999-2-004	V. Cortier, H. Ganzinger, F. Jacquemard, M. Veanes	Decidable fragments of simultaneous rigid reachability
MPI-I-1999-2-003	U. Waldmann	Cancellative Superposition Decides the Theory of Divisible Torsion-Free Abelian Groups
MPI-I-1999-2-001	W. Charatonik	Automata on DAG Representations of Finite Trees
MPI-I-1999-1-007	C. Burnikel, K. Mehlhorn, M. Seel	A simple way to recognize a correct Voronoi diagram of line segments
MPI-I-1999-1-006	M. Nissen	Integration of Graph Iterators into LEDA
MPI-I-1999-1-005	J.F. Sibeyn	Ultimate Parallel List Ranking ?
MPI-I-1999-1-004	M. Nissen, K. Weihe	How generic language extensions enable “open-world” desing in Java
MPI-I-1999-1-003	P. Sanders, S. Egner, J. Korst	Fast Concurrent Access to Parallel Disks
MPI-I-1999-1-002	N.P. Boghossian, O. Kohlbacher, H.-. Lenhof	BALL: Biochemical Algorithms Library
MPI-I-1999-1-001	A. Crauser, P. Ferragina	A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory
MPI-I-98-2-018	F. Eisenbrand	A Note on the Membership Problem for the First Elementary Closure of a Polyhedron
MPI-I-98-2-017	M. Tzakova, P. Blackburn	Hybridizing Concept Languages
MPI-I-98-2-014	Y. Gurevich, M. Veanes	Partisan Corroboration, and Shifted Pairing
MPI-I-98-2-013	H. Ganzinger, F. Jacquemard, M. Veanes	Rigid Reachability
MPI-I-98-2-012	G. Delzanno, A. Podelski	Model Checking Infinite-state Systems in CLP
MPI-I-98-2-011	A. Degtyarev, A. Voronkov	Equality Reasoning in Sequent-Based Calculi
MPI-I-98-2-010	S. Ramangalahy	Strategies for Conformance Testing
MPI-I-98-2-009	S. Vorobyov	The Undecidability of the First-Order Theories of One Step Rewriting in Linear Canonical Systems
MPI-I-98-2-008	S. Vorobyov	AE-Equational theory of context unification is Co-RE-Hard
MPI-I-98-2-007	S. Vorobyov	The Most Nonelementary Theory (A Direct Lower Bound Proof)
MPI-I-98-2-006	P. Blackburn, M. Tzakova	Hybrid Languages and Temporal Logic
MPI-I-98-2-005	M. Veanes	The Relation Between Second-Order Unification and Simultaneous Rigid E-Unification