

# Polymorphic Algorithms

## FFT-Implementations That Share

Marcin Zalewski and Sibylle Schupp

Dept. of Computer Science,  
Rensselaer Polytechnic Institute (RPI), Troy, NY  
{zalewm,schupp}@cs.rpi.edu

**Abstract.** We denote by a *polymorphic radix- $n$  FFT* an abstract algorithm scheme that is shared by all radix- $n$  FFT algorithms, similar to the way polymorphic data types share. Given such polymorphic algorithm, particular radix- $n$  algorithms can be obtained by specialization, thus need not be implemented separately. How to accomplish sharing between different radix- $n$  algorithms is not obvious: for example the four major radix-2 algorithms, defining different divide-and-conquer schemes and working for different input formats, have little in common at the implementation level. At a higher level of abstraction, however, it is possible to provide a unifying framework. In this paper we introduce a polymorphic radix- $n$  FFT, mathematically based on the Cooley-Tukey mapping, and show how to effectively realize this mapping using techniques from generic programming. Since specializations can take place entirely at compile time, their generalization does not incur any run-time overhead. We implemented the polymorphic radix- $n$  FFT as a C++ meta-program.

## 1 Introduction

An algorithm, as Knuth [12] defines it, is characterized by five properties: finiteness, definiteness, inputs (zero or more), outputs (one or more), and effectiveness. A *generic* algorithm, in difference, is an algorithm that lacks the property of definiteness. In a generic algorithm, program variables can be declared in terms of type parameters, not just types. Thus, the semantics of expressions and statements that make up the computational steps of an algorithm is not fully determined at algorithm design time. Collins therefore referred to generic algorithms as *algorithm schemes* [15].

Giving up definiteness is not simply a decision by an algorithm designer that is straightforward to realize. Instead, it requires a process of repeatedly revising interface and implementation of an algorithm and successively lifting restrictive assumptions. Much of the methodology of generic programming in fact is dedicated to the task of pushing abstractions further, both at the declarative level of data types and at the level of control structures. Although most of the widely used generic software, e.g., the Standard Template Library (STL), the Matrix Template Library (MTL), or the Boost Graph Library [18, 22, 14], is best known for its extensive utilization of polymorphic types (“templates”), of

equal importance are the conceptual abstractions and programming idioms that generalize control structures:

- *Iterators* allow abstracting from container types and traversing an iteration space in a generic manner.
- *Functors* support the idea of higher-order programming and, in the special case of predicates, determine the control flow in a generic way.
- *Adaptors* provide common interfaces for different implementations and types with different behavior.

Generic algorithms therefore always combine several kinds of abstractions simultaneously. A generic quicksort routine, for example, abstracts from the types of the elements to be sorted as well as from the sequence in which these elements are stored and the sorting key that is applied. Parameterized by a functor and an iterator, the generic quicksort thus generalizes the traditional quicksort to an algorithm scheme that works with (infinitely) many iterators, containers, and sorting keys. Since each non-generic instance is obtained merely by binding, or specializing, type parameters, and since these bindings take place at compile time, the generalization does not come at the price of efficiency.

Despite the far-reaching abstractions that now have become standard in generic programming, there seems to be a boundary that is not crossed: the boundary between different algorithmic realizations of a function and the function itself. While there are numerous polymorphic types, i.e., types that share their implementation, there is no polymorphic sorting function, for example, that could be specialized to different generic sorting algorithms; no framework that, if appropriately instantiated, turned into quicksort, but if instantiated differently became, say, mergesort. Instead, the different realizations of sorting have entirely separate implementations without sharing any code.

Is it impossible, then, to implement a *polymorphic* algorithm? Can only data types share, but function realizations cannot? In this paper we answer this question by giving an example. For the radix-2 Fast Fourier Transform (FFT) algorithms, the most important FFTs in practice, we present an algorithm scheme that contains the four commonly implemented radix-2 algorithms as specializations. These algorithms have sufficiently different type and control structures that, by looking at their implementation, it is not obvious at all how they could become instances of a common abstraction. The mathematical theory of the Cooley-Tukey mapping [3], however, provides a uniform theory for the different algorithms. Applying advanced techniques of generic programming, we are able to implement this mapping and to thereby provide a polymorphic radix-2 FFT. As a “side effect”, the polymorphic radix-2 algorithm becomes applicable to arbitrary radices, thus in fact is a polymorphic radix- $n$  FFT.

The paper is organized as follows. First, we recapitulate the different radix-2 algorithms. Next, we summarize the Cooley-Tukey mapping and highlight some design issues that need to be addressed if a symbolic formula should be mapped to an effective procedure. The polymorphic radix- $n$  algorithm itself is subject of sect. 4 where we discuss the underlying major design decisions first

language-independently and then with respect to C++, the implementation language chosen. An overview of related and future work and a brief discussion of possible other candidates of polymorphic algorithms complete the presentation. We assume readers to have a working knowledge of the FFT and the basics of the Standard Template Library (STL), and refer to the literature (e.g., [6, 2, 19, 18, 1]) otherwise.

## 2 Radix-2 FFT Algorithms

The Discrete Fourier Transform (DFT) is among the most fundamental operations in digital signal processing and many other application areas. Since the direct computation of a DFT requires  $\mathcal{O}(N^2)$  complex operations for input of length  $N$ , many methods were developed that compute DFTs more efficiently. All algorithms that achieve logarithmic behavior are collectively called Fast Fourier Transforms (FFTs). Although the history of FFTs can be traced back to Gauss [7], the short article by Cooley and Tukey [3] is often considered the beginning of the modern interest in FFT. Reducing the computation to  $\mathcal{O}(N \log_2 N)$  complex operations, their paper spawned considerable interest in the topic, which has not diminished since. In 1995, a bibliography [23] listed more than 3500 publications on FFTs and convolution algorithms.

In practice, *radix-2* FFT algorithms dominate. Although theoretically more efficient algorithms exist, most notably Winograd's algorithm [26], the split-radix algorithm [5], and the prime factor algorithm [10, 11, 21], radix-2 algorithms are most frequently implemented since their structure is simple and they allow for *in-place* transforms. In this section we briefly recapitulate the DFT and present the four basic in-place, radix-2 algorithms in more detail.

### 2.1 The Discrete Fourier Transform

In short, the DFT is a matrix-vector-product. Let  $(x_0, x_1, \dots, x_{N-1})^T$  be the input vector,  $(X_0, X_1, \dots, X_{N-1})^T$  the values of the transform, and  $W_N$  the  $N$ -th root of unity ( $W_N = e^{-i2\pi/N}$ ). The DFT can be written as

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ \vdots \\ X_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & W_N & W_N^2 & W_N^3 & \dots & W_N^{N-1} \\ 1 & W_N^2 & W_N^4 & W_N^6 & \dots & W_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \vdots & & \vdots \\ 1 & W_N^{N-1} & W_N^{2(N-1)} & W_N^{3(N-1)} & \dots & W_N^{(N-1)(N-1)} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{bmatrix} \quad (1)$$

or, more compactly, as

$$X_k = \sum_{i=0}^{N-1} x_i W_N^{ik}, \quad k = 0, \dots, N-1. \quad (2)$$

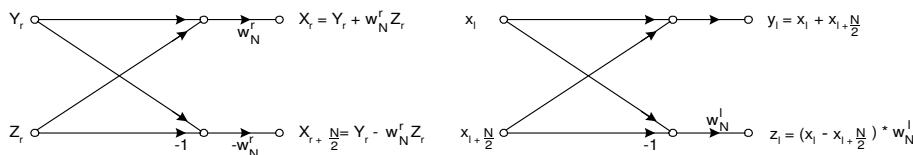
## 2.2 Decimation in Time and Frequency

All FFTs are divide-and-conquer algorithms, and they all exploit the special structure of the matrix in Eq. (1) to recursively decompose a transform into transforms of smaller length. How the decomposition is chosen, however, determines the characteristics of a particular FFT. For radix-2 algorithms, an  $N$ -point transform is decomposed into two transforms of length  $N/2$ .

One distinguishes two fundamental radix-2 decompositions: *decimation-in-time (DIT)* and *decimation-in-frequency (DIF)*. In DIT, subproblems are obtained by dividing the *input* series into an even-indexed set  $\{x_{2k} | k = 0, \dots, N/2 - 1\}$  and an odd-indexed set  $\{x_{2k+1} | k = 0, \dots, N/2 - 1\}$ , whereas in DIF, the subproblems result from decimating the *output* series into an even-indexed set  $\{X_{2k} | k = 0, \dots, N/2 - 1\}$  and an odd-indexed set  $\{X_{2k+1} | k = 0, \dots, N/2 - 1\}$ . What both decimation directions have in common, though, is their constant geometry: the order of the intermediate calculations and the exact location of their inputs and outputs both are known before the algorithm runs. The sequence of operations thus varies only with the size of the input, not with its values. This constant geometry, along with the regularity of a divide-and-conquer approach, allows performing the transform by combining simple building blocks: *butterflies*, so called because of the shape of their flow graph.

## 2.3 Butterflies

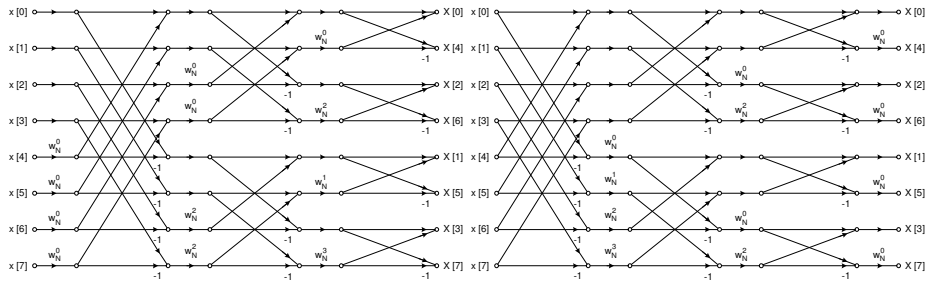
A butterfly maps a pair of values to another pair: it takes two intermediate values and forms two new values by adding and subtracting the input, and multiplying it with a corrective factor, the *twiddle factor*. The two kinds of decimation each define their own butterfly: DIT defines the Cooley-Tukey butterfly [3], DIF the Gentleman-Sande butterfly [8]. Despite their similar shape, the butterflies represent quite different arithmetical operations, as depicted by the annotated flow graphs in Fig. 1. Since monomorphic radix-2 algorithms are based on butterflies,



**Fig. 1.** Cooley-Tukey butterfly (left) and Gentleman-Sande butterfly (right)

the computations of DIT and DIF differ accordingly. In short, the DIT algorithm consists of three nested loops: the outermost loop, which controls the division into subsequences, the middle loop, which iterates over the resulting subsequences, and the innermost *butterfly-loop*, which performs  $n/2$  butterfly computations for a subproblem of size  $n$ . In DIT, the twiddle factors in the butterfly-loop are constant. The DIF algorithm, on the other hand, while consisting of the same loop

nest, splits the input sequence differently in its outermost loop. Moreover, the twiddle factors in its butterfly-loop depend on the loop index. Fig. 2 illustrates the different kinds of butterflies that are assembled.

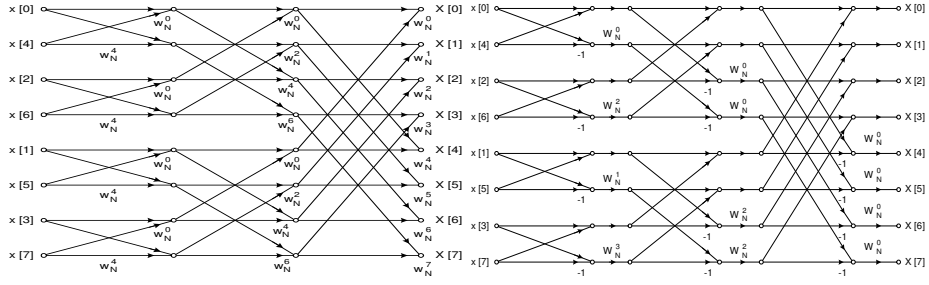


**Fig. 2.** Flow graph of the decimation-in-time (left) and the decimation-in-frequency (right) 8-point FFT. Redrawn from Oppenheim & Schaffer [20], Figs. 9.14, 9.20.

## 2.4 Natural and Bit-Reversed Output Order

Another element in the design of a radix-2 FFT is given by the order in which the output is computed. Since the algorithms are in-place, each butterfly overwrites two values of the input sequence. As a result, the output is permuted, albeit in a regular way: for radix 2 it is bit-reversed. If the output needs to be in natural order, it is easiest to bit-reverse the input. Again, the different decomposition directions behave differently (see Fig. 3). In decimation-in-time with bit-reversed input, the twiddle factors in the butterfly-loop depend on the loop index, while in decimation-in-frequency with bit-reversal, the twiddle number in the innermost loop is constant. At the same time, both algorithms differ from their counterpart for natural input with respect to the iteration space that the butterfly-loop defines. In the radix-2 algorithms with natural order, the butterfly-loop initially has length  $N/2$  and decreases by a factor of 2 in each iteration; in the radix-2 algorithms with bit-reversed input, the inner loop has initial length 1 and increases by a factor of 2.

In summary, the four major radix-2 algorithms can be distinguished according to their decimation direction and the order they assume for their input vector. As we have seen, they differ with respect to the iteration space, the loop variables, or both. No implementation we are aware of allows for any sharing between the implementation of any two of them: they constitute four entirely separate implementations. Relinquishing the use of the butterflies, however—the fundamental building blocks of current radix-2 implementations—gives way to a single, unifying framework.



**Fig. 3.** Flow graph of an 8-point FFT with bit-reversed input: decimation in time (left) and frequency (right). Redrawn from Oppenheim & Schaffer [20], Figs. 9.7, 9.22.

### 3 The Cooley-Tukey Mapping

The divide-and-conquer approach proposed by Cooley and Tukey [3], although often identified with a radix-2 FFT, can be applied to any DFT of length  $N$  provided  $N$  is composite. In this section we summarize the Cooley-mapping, following the tutorial by Duhamel and Vetterli [6], show how it specializes to the decompositions in DIF and DIT, respectively, and discuss fundamental design issues for an effective representation of the Cooley-Tukey mapping.

#### 3.1 The Mapping

Let us assume that the length of the transform is composite:  $N = N_1 \cdot N_2$ . It is fairly easy to see that Eq. (2) can be written as a product so that the second factor represents  $N_1$  DFTs of length  $N_2$ :

$$X_k = \sum_{n_1=0}^{N_1-1} W_N^{n_1 k} \sum_{n_2=0}^{N_2-1} x_{n_2 N_1 + n_1} W_{N_2}^{n_2 k}. \quad (3)$$

The full generality of the Cooley-Tukey mapping, however, comes from the fact that the outputs of these DFTs serve as input to another set of DFTs. To that end, denote by  $Y_{n_1, k}$  the  $k$ -th output of the  $n_1$ -th transform:

$$Y_{n_1, k} = \sum_{n_2=0}^{N_2-1} x_{n_2 N_1 + n_1} W_{N_2}^{n_2 k}. \quad (4)$$

Since  $Y_{n_1, k}$  does not depend on the index  $k$  ( $W_{N_2}^k = W_{N_2}^m$  for  $k = N_2 + m$ ), sum and product in Eq. (3) can be changed:

$$X_k = \sum_{n_1=0}^{N_1-1} Y_{n_1, k} W_N^{n_1 k}. \quad (5)$$

Expressing the index  $k$  as a congruence modulo  $N_2$ , next, allows factoring the twiddle numbers in Eq. (5) into a product with  $N_1$ -th twiddle numbers ( $W_{N_1}^k = W_N^{n_1(k_1 N_1 + k_2)} = W_N^{n_1 k_2} W_{N_1}^{n_1 k_1}$ ). Setting then

$$Y'_{n_1, k_2} = Y_{n_1, k_2} W_N^{n_1 k_2}, \quad (6)$$

we arrive at the final form of Eq. (5)

$$X_{k_1 N_2 + k_2} = \sum_{n_1=0}^{N_1-1} Y'_{n_1, k_2} W_{N_1}^{n_1 k_1}. \quad (7)$$

Summarizing Eqs. (3)-(7), one can identify three stages in the Cooley-Tukey mapping. First,  $N_1$  DFTs of length  $N_2$ , represented by  $Y_{n_1, k}$ , are computed. Second, the results of these  $N_1$  DFTs are multiplied by the twiddle factors in  $Y'_{n_1, k_2}$ . And third,  $N_2$  DFTs of length  $N_1$ , represented by the summation in Eq. (5), are performed on the results of the second step.

The Cooley-Tukey mapping describes only one stage of the divide-and-conquer method. If the lengths of the subproblems,  $N_1$  and  $N_2$ , are composite themselves, the mapping applies again. How the values for  $N_1$  and  $N_2$  are chosen in each step, determines the resulting FFT: to specialize Eq. (5) to a radix-2 DIT, one must choose  $N_1 = 2$  and  $N_2 = N/2$  in every step, to specialize it to a radix-2 DIF, on the other hand, one must set  $N_1 = N/2$  and  $N_2 = 2$ . If the divisor is not 2 but another factor  $n$  of  $N$ , the Cooley-Tukey mapping specializes to a radix- $n$  algorithm. Finally, if the choice of factors varies between different decomposition stages, so-called *mixed-radix* algorithms result. In any case, when no further decomposition is possible, efficient direct computation methods must be used. For example in a radix- $n$  computation, subproblems of size  $n$  are computed directly.

### 3.2 Design Issues

The formula in Eq. (7) provides the uniform framework required for a polymorphic radix- $n$  algorithm. However, how do we get from Eq. (7) to an effective procedure? Moreover, what additional issues have to be considered so that an instance of the polymorphic radix- $n$  algorithm is guaranteed to be no less efficient than its direct, monomorphic implementation?

Obviously, the decomposition factors  $N_1$  and  $N_2$ , since they control the specialization, can no longer be implicitly assumed in the algorithm. Instead, there has to be a way to bind them to different values. Moreover, their values need to be known statically since otherwise specializations cannot take place at compile time. Representing  $N_1$  and  $N_2$  as static parameters, however, raises the question how their decomposition is accomplished in the subsequent recursion stage and, more generally, how the divide-and-conquer scheme is defined both abstractly and statically. If users “select” the desired FFT via specialization at the top level of the polymorphic algorithm, its body needs to propagate their choice and to perform the appropriate recursion step then automatically. An abstract and

static divide-and-conquer scheme provided, next, the twiddle factors and the input sequence need to be modeled so that they allow for the different access patterns that the different decimations require.

For efficiency reasons, twiddle factors usually are pre-computed rather than calculated on the fly, and stored in a container. Depending on the decimation chosen, the step-size varies with which this sequence is traversed. Again, the polymorphic algorithm can no longer implicitly operate with a particular step-size but has to refer to it abstractly. Similarly with the input sequence, which is given in natural order in some cases, in bit-reversed order in others. Looping over the input sequence has to be expressed abstractly, so that one instance of the loop steps through the sequence in the “random” fashion dictated by the bit-reversal, while another instance traverses it linearly. Finally, the view on the input and the twiddle sequence need to be coupled so that they always assume the same input order. How to properly identify the required parameters and how to design the parameterization to allow for compile-time specialization, is subject of the next section.

## 4 A Polymorphic Radix- $n$ Algorithm

The algorithm we present follows closely the Cooley-Tukey formula, i.e., it solves a given problem of length  $N$  recursively, by dividing it into  $N_1$  subproblems of size  $N_2$  and  $N_2$  subproblems of size  $N_1$ , with  $N = N_1 \cdot N_2$ . The division stops at subproblems of size of the chosen radix and then applies a direct computation method. In the following, we refer to the sizes of the subproblems by  $N_1$  and  $N_2$ .

### 4.1 Data Abstraction

Essential for the intended code sharing, as we have argued in Sect. 3, is the ability to access data in a uniform and abstract manner, independently from the underlying container and the method of its traversal. The concept of an *iterator* proves to be the appropriate abstraction for this problem. An iterator adds a layer of abstraction between the data and its access by providing an interface between both. This interface specifies operations like read or write access to the current element the iterator is pointing to, advancement to the next element, or comparison with other iterators to test the relative position in the data range. How each of these operations is defined, however, is left unspecified. Different implementations of a particular iterator category, thus, can provide different traversal protocols while still complying to a common interface. Because of this indirection, iterators are widely used in generic programming and, through the Standard Template Library, standardized. In our algorithm, the iterator abstraction makes it in particular possible to abstract from the index-notation used in the mathematical representation of the Cooley-Tukey mapping. Abstracting from indices is necessary for a polymorphic algorithm, since the layout of the data may not correspond to the positions defined by an index.

Fig. 4 shows the structure of the body of the polymorphic radix- $n$  algorithm, shared among its different specializations: in direct correspondence to the mathematical Cooley-Tukey mapping, it is a sequence of three loops. The first loop performs the DFTs of size  $N_2$  computed in  $Y_{n_1,k}$  (Eq. (4)), the second loop defines the twiddle factor multiplication in  $Y'_{n_1,k_2}$  (Eq. (6)), and the last loop represents the DFTs of size  $N_1$  computed in the outermost summation in Eq. (7). The two program variables *first\_iterator* and *second\_iterator*, in the first and the last loop, encapsulate the different advancement semantics of DIT or DIF, with natural or bit-reversed input. The multiplication by twiddle factors, in the middle loop, requires a traversal protocol other than sequential advancement, since the access to twiddle factors changes with the loop iteration. Besides, a twiddle factor with index  $j * k$  is not necessarily located at the  $(j * k)^{th}$  position of the twiddle container. We use a generic index operation, usually referred to as random access operation, to hide the location of the proper twiddle factor.

```

1.      for i from 1 to  $N_1$ 
2.          perform DFT of size  $N_2$  on data beginning at first_iterator
3.          advance first_iterator to the next position
4.
5.      for j from 0 to  $(N_1 - 1)$ 
6.          for k from 0 to  $(N_2 - 1)$ 
7.              multiply  $k^{th}$  output from  $j^{th}$  DFT of size  $N_2$ 
                  by  $(j * k)^{th}$  twiddle_iterator
8.
9.      for i from 1 to  $N_2$ 
10.         perform DFT of size  $N_1$  on data beginning at second_iterator
11.         advance second_iterator to next position

```

**Fig. 4.** Basic loop structure of polymorphic radix- $n$  algorithm

Since  $N_1$  and  $N_2$  change in the course of the computation, the loops in Fig. 4 are not constant but change their iteration space from one recursion step to the next. Consequently, the behavior of the iterators has to be adjusted between the recursion steps, and this adjustment has to be done in a generic and statically decidable manner. In generic programming, *iterator adaptors* are used for that purpose, i.e., class templates that are parameterized by an iterator and modify or extend some of its functionality. Static adjustments of iterator properties, thus, can be achieved by type-based iterator adaptors, which modify the type of an iterator rather than a particular instance. The program variables *first\_iterator*, *second\_iterator*, and *twiddle\_factor*, therefore, may be iterator adaptors, and not necessarily iterators directly. In the actual implementation they adjust an iterator by modifying its stride, i.e., the step-size with which it traverses.

## 4.2 Static Programming

We motivated the need for static algorithm specialization already earlier. As it turns out, four static parameters suffice to deduce all other statically known values. Using the identifiers chosen in the pseudo-code listing (Figs. 5, 6), these are specified as follows:

- **Radix** is the radix of the algorithm and is necessary to test for the base case when **Length** is bound to the same value as **Radix**.
- **Length** is the length of the current problem and, together with **Radix**, determines  $N_1$  and  $N_2$ .
- **Decimation** is used to decide whether  $N_1 = \text{Radix}$  and  $N_2 = \text{Length}/\text{Radix}$  or vice versa; it thus defines whether the algorithm is DIT or DIF.
- **Order** corresponds to the (natural or bit-reversed) input order and is used to instantiate iterator adaptors, which then adjust the behavior of the input and twiddle iterators.

In addition, we assume a statically typed implementation language, so that the types of the dynamic parameters, the input and the twiddle iterator, can be used as a base for any recursive adjustments. The recursion of the divide-and-conquer scheme itself is trivial and only requires passing on to the subsequent subproblem the new length and, unchanged, the other three static parameters.

## 4.3 Specializations

In the previous two subsections we have seen that the polymorphic radix- $n$  FFT expresses the size of the FFT subproblems symbolically in terms of  $N_1$  and  $N_2$  and that it accesses input and twiddle data indirectly, through iterator adaptors. We pointed out that the size of a subproblem can be statically computed and explained that iterator adaptors have to be parameterized so that the underlying iterator can adjust e.g. its stride from one recursion step to the next. The complete pseudo-code is given in Sect. B in the appendix. In illustration, Table 1 lists for the four major radix-2 FFTs the specific bindings of  $N_1$ ,  $N_2$ , and the strides of the three iterator variables that control the loops in the body of the polymorphic algorithm (see Fig. 4).

**Table 1.** Recursive adjustments in the strides of the iterators for radix-2 FFTs

	$N_1$	$N_2$	first_iterator	second_iterator	twiddle_iterator
$\text{DIT}_{NR}$	2	$N/2$	–	old-stride $\cdot$ 2	–
$\text{DIT}_{RN}$	2	$N/2$	old-stride $\cdot$ 2	–	old-stride $\cdot$ 2
$\text{DIF}_{NR}$	$N/2$	2	–	old-stride $\cdot$ $N/2$	old-stride $\cdot$ 2
$\text{DIF}_{RN}$	$N/2$	2	old-stride $\cdot$ $N/2$	–	–

#### 4.4 Implementation

The polymorphic radix- $n$  algorithm is implemented in the C++ programming language. C++ is the most widely used language for generic programming since it supports parameterized types (“templates”) as well as static meta-programming. In our particular application three additional properties of the template type system are useful, which are worthwhile mentioning. First, template parameters can be not only type parameters, but also so-called *non-type template parameters*. In general, non-type parameters are (static) expressions; constants such as the radix or the length of the input sequence thus map directly to them. Second, the instantiation, hence specialization, of parameterized types is done automatically rather than by hand—a fact that meta-programming exploits. If, for example, the factors  $N_1, N_2$  of the input are represented as static *expressions* in the template parameter list of a class, the compiler is enforced to evaluate this expression when it needs to instantiate the enclosing template; the values of  $N_1, N_2$  can thus be automatically computed. Third, C++ unifies templates in a best-fit manner, based on a partial order of templates. Thus, if the Cooley-Tukey recursion is represented by a pair of class templates, for the general and the base case, respectively, the compiler automatically detects and selects the base case when appropriate. Shifting much of the work to the C++ template sys-

**Table 2.** Interfaces of the major class templates in the C++ implementation

Algorithms
template<int length, int radix, class Decimation, class Order> class radix_mapping; template<int fftlength> class small_dft; template<> class small_dft<2>; template<> class small_dft<3>; ...
Iterator Adaptors
template <class Iterator, int stride> class strided_iterator template<class Iterator, int offset, int next, int divider> class offset_iterator; template<class Iterator, int length> class bit_reversal_iterator;
Container Adaptors
template<class Storage, class TwiddleSet> class twiddle_container; template<class RandomAccessContainer, int length> class bit_reversal_container;

tem, the polymorphic radix- $n$  code becomes surprisingly slim. The core of the algorithm is formed by two class templates, `radix_mapping`, which implement the Cooley-Tukey mapping; one class organizes the general case of the recursion, the other one the base case. In the latter case, the computation is forwarded to a direct computation of the FFT, encapsulated in the class template `small_dft`. There exists one `small_dft` per base case, e.g., `small_dft<2>` for radix-2 algorithms. The classes for the polymorphic algorithm are complemented by a set of iterator adaptors. We provide an iterator adaptor to change the stride, another one to change the offset, and a third one for bit-reversed traversal; as an implementation detail we mention that each of these iterators can be imple-

mented very compactly using so-called iterator policies and `iterator_adaptor` templates, provided elsewhere [4]. The last group of classes, finally, comprises container adaptors, in essence the class template `twiddle_container`, which allows storing (pre-computed) twiddle numbers according to different memory models, and the container adaptor `bit_reversal_container`, which permutes the input sequence as it is required for FFTs with natural output. The interfaces of the major classes (slightly simplified) are listed in Table 2; the code of the core class, `radix_mapping`, is completely listed in Sect. A of the appendix.

## 5 Related Work

The work presented in this paper is rooted in the discipline of generic programming. Theoretically, the paper by Musser and Stepanov [16] was the first that postulated an “algorithm-oriented” approach and showed how to maximize the re-usability of an algorithm by reducing it to its minimal requirements. With the success of the Standard Template Library (STL), numerous successor libraries [4] have implemented generic algorithm schemes, which parameterize data types as well as the underlying implementation. Applying idioms and techniques like iterators and adaptors, our work, on the one hand, benefits from the knowledge now available for generic programming. On the other hand, giving an example of different (radix- $n$ ) algorithms that share their implementation much like polymorphic data types do, pushes algorithm abstraction yet a little further.

As we showed, crucial for the design and efficiency of our polymorphic radix- $n$  FFT is the identification of the right abstractions and their representation as statically bound-able parameters. Methodologically speaking, our work is therefore closely related to the work by Lawall [13] and Glück et. al. [9], who both investigate binding-time analysis in the context of numerical algorithms. Lawall, interested in the differences between compile-time and run-time program specialization as well as their dependencies, shows how to significantly improve an unoptimized FFT by a specialization that then allows for loop unrolling and certain compile-time computations. Glück et. al. take a somewhat broader view by classifying numerical computations according to their degree of data dependence. From there, they demonstrate and derive to what extent binding-time analysis can identify sources of specialization. What both papers have in common—and what distinguishes them from ours—is their purpose to apply binding-time analysis for the specialization and then optimization of a program. In contrast, we use binding-time analysis for the purpose of generalization, and we specialize to definite algorithms rather than to more efficient ones.

Another difference is that both Lawall and Glück use special partial evaluator tools while we realize partial evaluation within the C++-implementation language, through a meta-program. One of the first to see the connection between partial evaluation and the C++ template sub-language was Todd Veldhuizen [24, 25], who demonstrated how compile-time recursion is possible if template parameters are appropriately chosen; today, it is known that the template sub-language is Turing-complete. Incidentally, Veldhuizen also used the FFT to

illustrate meta-programming, but, in difference to our goal, has not attempted to build an entire FFT library.

## 6 Conclusion and Future Work

We have presented a polymorphic FFT algorithm, based on the Cooley-Tukey mapping. While (non-generic) radix- $n$  FFTs, due to their different function bodies and their entirely different flow of control, commonly require different implementations, the high degree of abstraction of the polymorphic radix- $n$  FFT provides a common framework where each particular radix- $n$  FFT can be obtained merely by parameter specialization. As we showed, such generalization need not incur any overhead in run time or space: in our C++ implementation, all relevant parameters are static parameters so that the dispatch to a particular, non-generic radix- $n$  algorithm happens at compile time.

We identified three tasks for the immediate future. First, we want to further generalize the current algorithm from a polymorphic radix- $n$  to a polymorphic *mixed-radix* and even polymorphic *split-radix* algorithm. The former, mixed-radix scheme allows users to choose different radices at different recursion steps; its realization essentially requires a different instantiation of the current parameter for the value of the radix and should otherwise be fairly simple. The latter, split-radix method, dividing an FFT computation in sub-computations of different sizes, changes the nature of the recursion and the declarative structure of the current radix- $n$  algorithm. It furthermore requires a way to associate different recursion bases to the sub-computations of different sizes. However, since the current program already uses meta-programming techniques, it should be possible to implement the necessary modifications as an extension. Second, we need to be able to handle input of arbitrary size. Since the Cooley-Tukey mapping assumes input sizes of composite numbers, the current FFT algorithm does not work with (most) input sizes that are prime numbers. Moreover, no factors of the input size  $N$  can be larger than 21, since no efficient method is known to directly compute an FFT of size greater than 21. The prime factor algorithm [10, 11, 21], on the other hand, can handle arbitrary input sizes provided they are prime or their factors are relatively prime. We are therefore planning on completing the current radix- $n$  algorithm by an implementation of the prime factor algorithm. Third, we have to analyze the performance behavior of the actual instances of the polymorphic radix- $n$  algorithm. Although the parameter specialization happens statically, zero overhead in theory does not mean zero overhead in practice. In preliminary experiments we observed extreme variations (up to a factor of 1000!) and strange non-monotonic effects in the run time of an FFT-instance, depending on the compiler as well as the architecture. At the moment, we are speculating that bad cache behavior is to blame, caused perhaps by compile-time recursion, but we need to take into account also other factors such as inlining, template code bloat, and register pressure. Given a better understanding of the performance behavior, we will be able to tweak the code accordingly and to in-

clude specific optimizations where necessary. Our ultimate goal is to submit the FFT framework to Boost [4], the largest library initiative in C++.

A separate topic, and interesting on its own, is the question how the idea of polymorphism carries over to other families of algorithms. A proper “structural” view provided, we suspect to be able to polymorphically express other algorithms as well. For example, it seems to be possible to provide a common framework for the quadratic sorting algorithms selection, insertion, and bubble sort. Such framework would abstract from the different termination conditions in the two nested loops that characterize these algorithms; because of their dependency on run-time values, the resulting polymorphism is less statically here than in case of the FFT. It might also be possible to unify quicksort and introsort [17], if the transfer to introsort’s “stopper” can be encapsulated, e.g., in a functor. A third set of candidates form various multiplication algorithms, e.g. matrix-matrix or polynomial multiplications; the latter, being a convolution, is already closely related to the FFT scheme. While we certainly do not claim that every algorithm can be “made polymorphic”—the same does not hold true for data types either—or that it would even be desirable to unify algorithms of different complexity or with different functional specification, implementors of software libraries and reusable software in general could benefit from sharing at algorithmic level, whenever polymorphism is appropriate.

**Acknowledgment** R. Loos was the first to point out to us the difference between an algorithm and a generic algorithm and who made us aware of Collin’s notion of an algorithm scheme.

## References

1. M. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999.
2. E. Chu and A. George. *Inside the FFT Black Box. Serial and Parallel Fast Fourier Transform Algorithms*. CRC Press LLC, 2000.
3. J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, April 1965.
4. B. Dawes and D. Abrahams. Boost. <http://www.boost.org>.
5. P. Duhamel and H. Hollmann. ‘Split radix’ FFT algorithm. *Electronics Letters*, 20(1):14–16, January 1984.
6. P. Duhamel and M. Vetterli. Fast Fourier transforms: A tutorial review and a state of the art. *Signal Processing*, 19:259–299, 1990.
7. C. F. Gauss. *Werke III, Nachlass*, chapter Theoria interpolationis methodo nova tractata, pages 265–330. Reprint by G. Olms Verlag, Hildesheim, New York, 1886.
8. W. M. Gentleman and G. Sande. Fast Fourier transforms—for fun and profit. In *1966 Fall Joint Computer Conf., AFIPS Proc.*, pages 563–578, 1966.
9. R. Glück, R. Nakashige, and R. Zöchling. Binding-time analysis applied to mathematical algorithms. In J. Dolezal and J. Fidler, editors, *System Modelling and Optimization*, pages 137–146. Chapman & Hall, 1995.
10. I. J. Good. The interaction algorithm and practical Fourier analysis 1. *J. Roy. Statist. Soc. Ser. B*, B-20:361–372, 1958.

11. I. J. Good. The interaction algorithm and practical Fourier analysis 2. *J. Roy. Statist. Soc. Ser. B*, B-22:372–375, 1960.
12. D. E. Knuth. *The Art of Programming*, volume 1. Addison-Wesley, 1997.
13. J. Lawall. Faster Fourier transforms via automatic program specialization. In J. Hatcliff, T. Mogensen, and P. Thiemann, editors, *Partial Evaluation: Practice and Theory. Proc. of the 1998 DIKU Internat. Summerschool*, volume 1706, pages 338–355. Springer-Verlag, 1999.
14. L.-Q. Lee, J. Siek, and A. Lumsdaine. The generic graph component library. In *Proc. of the 1999 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, pages 399–414, 1999.
15. D. Musser, S. Schupp, and R. Loos. Requirements-oriented programming. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming—International Seminar, Dagstuhl Castle, Germany 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, Germany, 2000.
16. D. Musser and A. Stepanov. Algorithm-oriented generic libraries. *Software-practice and experience*, 27(7):623–642, Jul 1994.
17. D. R. Musser. Introspective sorting and selection algorithms. *Software-Practice and Experience*, 27(8):983–993, 1997.
18. D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library*. Addison Wesley, 2nd edition, 2001.
19. H. J. Nussbaumer. *Fast Fourier Transforms and Convolution Algorithms*. Springer Verlag, 1982.
20. A. V. Oppenheim and R. W. Schaffer. *Discrete-Time Signal Processing*. Prentice Hall Signal Processing Series, 2nd edition, March 1989.
21. C. M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proc. IEEE*, 56:1107–1008, 1968.
22. J. G. Siek and A. 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. H. V. Sorensen, C. S. Burrus, and M. T. Heideman. *Fast Fourier Transform Database*. PWS Publishing Co., Boston, MA 02116-4324, 1995.
24. T. Veldhuizen. Using C++ template metaprograms. In S. Lippman, editor, *C++ Gems*, pages 36–43. SIGS, New York, May 1995.
25. T. L. Veldhuizen. C++ templates as partial evaluation. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 13–18, 1999.
26. S. Winograd. On computing the discrete Fourier transform. *Proc. of the Nat. Academy of Sciences of the United States of America*, 73(4):1005–1006, April 1976.

## A C++-Interface of the Cooley-Tukey Mapping

```

// General radix_mapping stage
template<int length,      // length of the DFT
        int radix,      // desired radix, must have a small DFT
                          // implemented
        typename Decimation, // DIT or DIF
        typename Order,    // input order: natural to bit-reversed
        typename Tag,      // reserved for future use
        bool verbose>     // debugging switch

```

```

class radix_mapping {
public:
    typedef radix_mapping<length,radix,Decimation,Order,Tag,
        verbose> self;

    // Factor 'length' into N1 and N2, based on 'Decimation'
    typedef typename
        if_types<equal_types<Decimation,radix_dit>::equal,
            STATIC_INT<radix>,
            STATIC_INT<length / radix> >::if_t    N1_t;
    typedef typename
        if_types<equal_types<Decimation,radix_dit>::equal,
            STATIC_INT<length / radix>,
            STATIC_INT<radix> >::if_t          N2_t;
    enum { N1 = N1_t::value};
    enum { N2 = N2_t::value};

    // Define the type for DFTs of length N1
    typedef radix_mapping<N1,radix,Decimation,Order,Tag,verbose>
        stageN1;
    // Define the type for DFTs of length N2
    typedef radix_mapping<N2,radix,Decimation,Order,Tag,verbose>
        stageN2;

    template<class RandomAccessIterator, class TwiddleIterator>
    radix_mapping(const RandomAccessIterator& input_iterator,
        const TwiddleIterator& twiddle_iterator)
    {

        // Static computation:
        // Adjust access patterns of the input and twiddle iterator
        // Adjust by N2
        typedef typename adjusted_boost_iterator_trait<
            RandomAccessIterator,Order,STATIC_INT<N2> >
            ::iterator          first_iterator;
        // Adjust by N1
        typedef typename adjusted_boost_iterator_trait<
            RandomAccessIterator,
            typename inverted_order_trait<Order>::order,STATIC_INT<N1> >
            ::iterator          second_iterator;

        typedef typename adjusted_boost_iterator_trait<
            TwiddleIterator,
            radix_mapping_reversed,STATIC_INT<radix> >
            ::iterator          twiddle_iterator_adj;

        // Dynamic computation:
        // Copy the original iterator
        first_iterator in_it(input_iterator.base());
        stageN2 s(second_iterator(in_it.base()),

```

```

        twiddle_iterator_adj(twiddle_iterator.base());
in_it++;

// Perform N1 stages of length N2
for(int i = 1; i < N1; in_it++, i++) {
    stageN2 s(second_iterator(in_it.base()),
              twiddle_iterator_adj(twiddle_iterator.base()));
    second_iterator second(in_it.base());
    second++;
    for(int k2 = 1; k2 < N2; k2++, second++) {
        *second *= twiddle_iterator[i * k2];
    }
}

// Copy the original iterator; has to be a strided iterator
// since stages in the second loop start at positions k2*N1
// where k2 = 0, 1, ..., N2-1
second_iterator in_it2(input_iterator.base());

// Perform N2 DFTs of length N1
for(int i = 0; i < N2; in_it2++, i++) {
    stageN1 s(first_iterator(in_it2.base()),
              twiddle_iterator_adj(twiddle_iterator.base())); }
};

```

## B The Polymorphic Radix- $n$ Algorithm (Pseudo-code)

In this section, the operator  $\leftarrow_P$  denotes static evaluation, followed by a static binding of the resulting value, the operator  $\leftarrow$  simply static binding. Static parameters are enclosed in angle brackets  $\langle \dots \rangle$ , dynamic parameters in the usual parentheses  $(\dots)$ . The operator  $++$  refers to iterator advancement,  $:=$  to an assignment.

1.     **Radix- $n$**  (input\_iterator, twiddle\_iterator)
2.     **with static parameters:**
3.         Decimation     *DIF or DIT*
4.         Order           *Natural-bitreversed or bitreversed-natural*
5.         Length          *Length of the DFT s.t. Length = Radix*
6.         Radix           *Radix of the algorithm*
7.     **with dynamic parameters:**
8.         input\_iterator
9.         twiddle\_iterator
10.    **static computation:**
11.         *For small values, compute FFT directly*
12.         Dft  $\leftarrow$  Small\_Dft( $\langle$ Radix $\rangle$ )
13.    **dynamic computation:**
14.         Dft(input\_iterator) ■

**Fig. 5.** The polymorphic radix- $n$  algorithm (base case)

```

1. Radix-n (input_iterator, twiddle_iterator)
2. with static parameters:
3.   Decimation      DIF or DIT
4.   Order           Natural-bitreversed or bitreversed-natural
5.   Length          Length of the DFT
6.   Radix           Radix of the algorithm
7. with dynamic parameters:
8.   input_iterator
9.   twiddle_iterator

10. static computation:
11.   N1  $\leftarrow$  Decimation(Length, Radix)   Generate N1
12.   N2  $\leftarrow$  Decimation(Length, Radix)   Generate N2
13.   Generate iterator properties for stages of length N1 and N2
14.   StageN1_input_properties  $\leftarrow$  Order(N1, Input_iterator_properties)
15.   StageN2_input_properties  $\leftarrow$  Order(N2, Input_iterator_properties)
16.   Identify subproblems and bind to names
17.   StageN1  $\leftarrow$  Radix-n(Decimation, Order, Length  $\leftarrow$  N1, Radix)
18.   StageN2  $\leftarrow$  Radix-n(Decimation, Order, Length  $\leftarrow$  N2, Radix)
19.   Generate new twiddle iterator properties from the old ones and radix
20.   New_t_it_properties  $\leftarrow$  Adjust(Twiddle_iterator_properties, Radix)

21. dynamic computation:
22.   Construct new twiddle iterator properties from the previous ones
23.   new_twiddle_iterator  $\leftarrow$  twiddle_iterator with New_t_it_properties
24.
25.   first_iterator := input_iterator with StageN1_input_properties
26.   Calculate N1 DFTs of size N2
27.   for i from 1 to N1
28.     StageN2(first_iterator, new_twiddle_iterator)
29.     first_iterator++   Advance iterator
30.
31.   first_iterator := input_iterator with StageN1_input_properties
32.   for n from 0 to (N1 - 1)
33.     second_iterator := input_iterator with StageN2_input_properties
34.     for k from 0 to (N2 - 1)
35.       Multiply the outputs from N2-sized DFTs by twiddle factors
36.       second_iterator :=  $\uparrow$ second_iterator * twiddle_iterator[n * k]
37.       second_iterator++   Advance iterator
38.     first_iterator++   Advance iterator
39.
40.   second_iterator := input_iterator with StageN1_input_properties
41.   Calculate N2 DFTs of size N1
42.   for i from 1 to N2
43.     StageN1(second_iterator, new_twiddle_iterator)
44.     second_iterator++   Advance iterator ■

```

**Fig. 6.** The polymorphic radix- $n$  algorithm (general case)