The Advanced FFT Program Generator GENFFT

Matteo Frigo[†]
Stefan Kral[*]

[†]Vanu Inc., Cambridge, MA 02140, USA
E-Mail: athena@fftw.org

[*]Institute for Applied and Numerical Mathematics
Technical University of Vienna
Wiedner Hauptstrasse 8-10, A-1040 Wien, Austria
E-Mail: e9625239@student.tuwien.ac.at

The work described in this report was supported by the Special Research Program SFB F011 "AURORA" of the Austrian Science Fund FWF.

Abstract

`genfft` is a special purpose compiler that generates fast fragments of C code that compute the discrete Fourier transform (DFT). `genfft` produces 95% of the code of FFTW 2.1.2, one of the most efficient DFT libraries.

This report documents the implementation of `genfft`. We first illustrate the operation of `genfft` by means of two examples. We then document the details of the modules that comprise `genfft`.

Contents

Chapter 1

Introduction

This report describes `genfft`, a special purpose compiler that generates fast fragments of C code that compute the discrete Fourier transform (DFT). A high-level description of `genfft`'s operation is given in the paper [Fri99], written by one of the authors of the present report. This report instead focuses on the details of the implementation of `genfft`, and it is meant for researchers who are interested in the internal mechanisms of `genfft` or wish to modify `genfft` to carry out experiments with FFT algorithms and implementations.

`genfft` is used by FFTW [FJ, FJ98], a comprehensive collection of fast C routines for computing the discrete Fourier transform in one or more dimensions, of both real and complex data, and of arbitrary input size. FFTW is one of the fastest DFT programs currently available (see [ABF+99]) because of two unique features. First, FFTW automatically adapts itself to the underlying hardware. Second, the "inner loop" of FFTW (which amounts to 95% of the total code) is generated and optimized automatically by `genfft`.

The FFTW library is structured as a collection of *codelets*—straight-line sequences of C code. Codelets can be composed in many ways to yield different algorithms for computing the DFT. In FFTW's lingo, a specific composition of codelets is called a *plan*, which consists of data structures that dictate which codelets are to be executed in what order. The precise plan used by FFTW depends on the size of the input (where "the input" is an array of complex numbers), and on which codelets happen to run fast on the underlying hardware. The users need not choose a plan by hand, however, because FFTW chooses the fastest plan automatically. The mechanisms required for making this choice are described in [FJ97, FJ98].

`genfft` is a special purpose compiler that generates codelets. A special purpose compiler is necessary in the FFTW system for the following reason. Almost all of the execution time of FFTW is spent running codelets. Moreover, FFTW works best if the space of possible plans—FFT algorithms—is large. Consequently, FFTW needs a large number of optimized codelets. For example, the release 2.1.2 of FFTW comprises 120 codelets summing up to 56,000 lines of optimized C code. Writing this code by hand would be extremely time-consuming (if at all possible).

`genfft` is an unusual compiler, however. While a normal compiler accepts code written in some high-level programming language and outputs machine code, `genfft` inputs a single integer (the size of the transform) and outputs C code that computes a DFT of that size. Nonetheless, `genfft` is internally very similar to

Module	*Class*	*Section*	*Size*	*Purpose*
`Genfft`	core	3.1	656 lines	Program entry point
`Fft`	core	3.2	311 lines	DFT code generator
`Exprdag`	core	3.3	937 lines	Expression simplification
`Schedule`	core	3.4	171 lines	DAG partitioning
`Asched`	core	3.5	171 lines	DAG flattening
`To_c`	core	3.6	319 lines	Unparsing to C code
`Complex`	auxiliary	4.1	122 lines	Operations on complex numbers
`Dag`	auxiliary	4.2	108 lines	Dag manipulation routines
`Expr`	auxiliary	4.3	42 lines	Representation of expressions
`Magic`	auxiliary	4.4	61 lines	Global parameters
`Number`	auxiliary	4.5	152 lines	Arbitrary precision arithmetic
`Symmetry`	auxiliary	4.6	314 lines	Data symmetries
`Twiddle`	auxiliary	4.7	117 lines	Twiddle factor loading policies
`Util`	auxiliary	4.8	153 lines	Some generic functions
`Variable`	auxiliary	4.9	254 lines	Variable handling, unparsing support

Table 1.1: List of all `genfft` modules. For each module, we list the module name, its class (whether it is core or auxiliary), a reference to the respective section of this report describing it in detail, an estimate of the module size, and a short description of the module's purpose.

a compiler, because it creates a representation of the codelet in an intermediate language, it rewrites the representation to obtain a "better" one, it performs common-subexpression elimination, and it has its own scheduler. (Experimental versions of `genfft` also perform register allocation and low-level instruction scheduling.) Because it is special purpose, `genfft` performs certain optimizations that are advantageous for DFT programs but not appropriate for a general purpose compiler. Conversely, it does not perform optimizations that are not required for the DFT programs it generates (for example, loop unrolling).

`genfft` is distributed as part of FFTW and is freely available from the web site `http://www.fftw.org`. This report documents version 2.1.2 of the program.[1]

`genfft` is written in Objective Caml [Ler98], a functional language similar to ML [MTH90]. `genfft` consists of 15 modules, which we classify as either "core" or "auxiliary" in this report. Core modules implement the important algorithms of `genfft`—for example, the `genfft` scheduler. Core modules are documented in Chapter 3. Auxiliary modules implement book-keeping functionality that is likely to be less interesting to the reader—for example, the representation of complex numbers. Auxiliary modules are described in Chapter 4. Table 1.1 summarizes the modules of `genfft`.

`genfft` defines a handful of data types upon which most code operates. For reference purposes, Table 1.2 summarizes the data types defined by `genfft`.

[1] Version 2.1.3 is available and it does not introduce any substantial change.

Name	*Section*	*Purpose*
`Complex.expr`	4.1	Complex expressions
`Complex.variable`	4.1	Complex variables
`Dag.dag`	4.2	Stores dependency between instructions
`Dag.dagnode`	4.2	Node of a dag
`Dag.color`	4.2	"Color" attribute of dag nodes
`Expr.expr`	4.3	Symbolic arithmetic expressions
`Number.number`	4.5	Arbitrary-precision abstract number type
`Symmetry.symmetry`	4.6	Input, intermediate, and output symmetries
`Variable.array`	4.9	Name of an array
`Variable.variable`	4.9	Support for references and temporary variables

Table 1.2: List of all data types defined in `genfft`. For each data type we list its name (including a module prefix), a reference to the section of this report describing the data type in detail, and its purpose.

Synopsis. The rest of this report is organized as follows. Chapter 2 illustrates the operation of `genfft` by means of two examples. Chapter 3 documents the six core modules `Genfft`, `Fft`, `Exprdag`, `Schedule`, `Asched`, and `To_c`. Chapter 4 documents the nine auxiliary modules `Complex`, `Dag`, `Expr`, `Magic`, `Number`, `Symmetry`, `Twiddle`, `Util`, and `Variable`. Chapter 5 provides supplemental information about `genfft`.

Chapter 2

Operation of GENFFT

In this chapter, we discuss the high-level operation of `genfft` and we illustrate it by means of two examples. `genfft` operates in four phases: creation, simplification, scheduling, and unparsing. Section 2.1 describes these four phases. Section 2.2 shows how these four phases are applied to the generation of a C program that computes the complex DFT of size 4. Section 2.3 does the same for the real DFT of size 6.

2.1 Overview of GENFFT's Operation

When invoked with the command line

```
genfft -notwiddle <n>
```

`genfft` generates C code (a codelet) that computes the discrete Fourier transform of size n. The generation process consists of four phases: creation, simplification, scheduling, and unparsing.

Creation. In this phase, `genfft` produces a directed acyclic graph (*dag*) that encodes a DFT algorithm for a transform of size n. `genfft` implements several algorithms discovered in the past 35 years, and it employs the most appropriate for the given size. Specifically, the algorithm used is:

- The split-radix algorithm [DV90], if n is a multiple of 4.
- A prime factor algorithm (as described in [OS89, page 619]), if n factors into $n_1 n_2$, where $n_i \neq 1$ and $\gcd(n_1, n_2) = 1$.
- The Cooley-Tukey FFT algorithm [CT65] if n factors into $n_1 n_2$ where $n_i \neq 1$.
- (n is a prime number) Rader's algorithm for transforms of prime length [Rad68] if $n = 5$ or $n \geq 13$.
- A direct application of the following definition of the DFT

$$Y[i] = \sum_{j=0}^{n-1} X[j]\omega_n^{-ij} \ , \tag{2.1}$$

where ω_n denotes $e^{2\pi\sqrt{-1}/n}$, i. e., the primitive n-th root of unity. Both X and Y are vectors of n complex numbers.

(Equation (2.1) defines the forward DFT. The backward DFT uses ω_n^{ij} instead of ω_n^{-ij}.)

Simplification. In this phase, `genfft` reduces the number of arithmetic operations of the dag by applying local rewriting rules to each node. This phase also applies some nonlocal transformations, for example, common-subexpression elimination and other transformations that are specific to the DFT. Moreover, besides noticing common subexpressions, the simplifier also attempts to create them.

Scheduling. In this phase, `genfft` produces a topological sort of the dag (a "schedule"). This schedule is *cache oblivious* [FLPR99]. For transforms of size 2^k, the schedule minimizes the asymptotic number of register spills, no matter how many registers the target machine has. (See [FLPR99].) For other sizes, the schedule is no longer provably optimal, but it still works well in practice.

Unparsing. In this phase, the schedule is unparsed to produce C code.

2.2 Example 1: Four Point Complex DFT

This section shows a complete example of the operation of `genfft`. Specifically, we discuss the generation of C code for a 4-point complex DFT (without twiddle factors). In this section, we do not show the internal `genfft` representation of a codelet, but we show equivalent pseudo code for readability.

Section 2.2.1 describes the creation phase. Section 2.2.2 describes the simplification phase. Section 2.2.3 describes the scheduling phase. Section 2.2.4 discusses the final unparse to C.

2.2.1 The Creation Phase

When invoked with the command line `genfft -notwiddle 4`, `genfft` produces a directed acyclic graph (dag) that represents an algorithm for computing a DFT of size 4. A pseudo-code representation of the dag is shown in Table 2.1.

At this stage, the dag does not contain trivial operations, i. e., multiplications by 0 or 1 and additions of 0, because the creation phase removes them immediately to speed up the generation process. Other than that, the dag is not optimized at all. As you can see in the figure, for example, the common subexpression `Re(In[1]) + Re(In[3])` appears twice, and other common subexpressions exist.

```
Re(Out[0]) := (Re(In[0]) + Re(In[2])) + (Re(In[1]) + Re(In[3]))
Im(Out[0]) := (Im(In[0]) + Im(In[2])) + (Im(In[1]) + Im(In[3]))
Re(Out[1]) := (Re(In[0]) - Re(In[2])) + (Im(In[1]) - Im(In[3]))
Im(Out[1]) := (Im(In[0]) - Im(In[2])) - (Re(In[1]) + Re(In[3]))
Re(Out[2]) := (Re(In[0]) + Re(In[2])) - (Re(In[1]) - Re(In[3]))
Im(Out[2]) := (Im(In[0]) + Im(In[2])) - (Im(In[1]) - Im(In[3]))
Re(Out[3]) := (Re(In[0]) - Re(In[2])) - (Im(In[1]) + Im(In[3]))
Im(Out[3]) := (Im(In[0]) - Im(In[2])) + (Re(In[1]) - Re(In[3]))
```

Table 2.1: The result of the creation phase for a 4-point complex DFT. For a DFT of size $n = 4$, `genfft` uses the Cooley-Tukey algorithm. The algorithm decomposes the original problem into four subproblems of size $n = 2$. These smaller DFTs are calculated by the direct application formula (2.1). The operator `:=` denotes assignment of a variable.

2.2.2 The Simplification Phase

The simplifier performs two main tasks: it reduces the arithmetic complexity of the dag, and it collects common subexpressions. The DFT of size 4 in the current example does not offer any chance for arithmetic simplification, but it does contain some redundant subexpressions.

The output of this phase is a list of assignments such as the one shown in Figure 2.1. In the figure, we see that `genfft` introduces temporary variables to hold the values of common subexpressions. The assignment list in the figure should not be interpreted as executable, however, because the order of assignments is arbitrary. (In particular, a value may be used before being computed.) The assignment list must be viewed as a dependency dag such as the one shown in the right-hand side of the figure.

2.2.3 The Scheduling Phase

Interpreting the assignment list as a dag, the scheduler produces a total order of the dag that can be executed by a sequential processor. The scheduling algorithm was designed so as to minimize the the lifetime of variables in the resulting code.

The scheduling phase further consists of two parts, unimaginatively called "sched" and "asched" in the `genfft` source code.

The *sched* phase transform the dag into a series-parallel dag. (Series-parallel dags [Val78] are a straightforward extension of the notion of series-parallel graphs [Duf65, Mac92, RS42].) The `genfft` source code calls the output of this step a *schedule*. This terminology is inappropriate because the total order is not completely specified yet, but we shall stick to it for consistency with the source code. A schedule (a series-parallel dag) is a recursive composition of series-parallel subdags. Subdags can be composed either serially or in parallel. In a serial composition, one subdag is executed completely before the other subdag begins

```
1   t1 := Re(In[0])
2   t2 := Re(In[2])
3   t3 := t1+t2
4   t4 := Re(In[1])
5   t5 := Re(In[3])
6   t6 := t4+t5
7   Re(Out[2]) := t3-t6
8   Re(Out[0]) := t3+t6
9   t7 := Im(In[0])
10  t8 := Im(In[2])
11  t9 := t7-t8
12  t10 := t4-t5
13  Im(Out[1]) := t9-t10
14  Im(Out[3]) := t10+t9
15  t11 := t1-t2
16  t12 := Im(In[1])
17  t13 := Im(In[3])
18  t14 := t12-t13
19  Re(Out[3]) := t11-t14
20  Re(Out[1]) := t11+t14
21  t15 := t7+t8
22  t16 := t12+t13
23  Im(Out[2]) := t15-t16
24  Im(Out[0]) := t15+t16
```

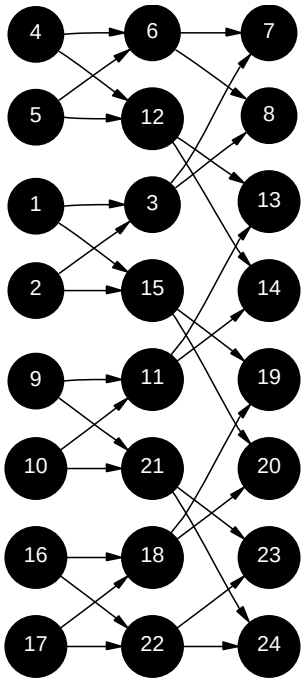

Figure 2.1: The result of the simplifier phase. The left-hand side shows an assignment list. The right-hand side shows the the corresponding dag. The node labels of the dag refer to line numbers on the left-hand side of the figure. Edges in the graph indicate data dependencies. An arrow from node a to node b indicates that b depends on a.

execution. In a parallel composition, the relative execution order of subdags is not specified.

In the *asched* phase, the schedule is transformed into an *annotated schedule*. The schedule annotator performs the following three tasks.

- The annotator imposes a serial order onto parallel blocks of the series-parallel dag. If two parallel blocks use mostly the same variables, then the annotator attempts to schedule them one after the other, so as to minimize the variable lifetime.

- For each variable, the annotator finds the smallest subdag that encompasses the entire lifespan of the variable.

- Finally, the annotator interprets the set of subdags as a nested set of C blocks in a "reasonable" way. By "reasonable" we mean that a block should not be too small (i.e., it should not contain only one instruction) nor too

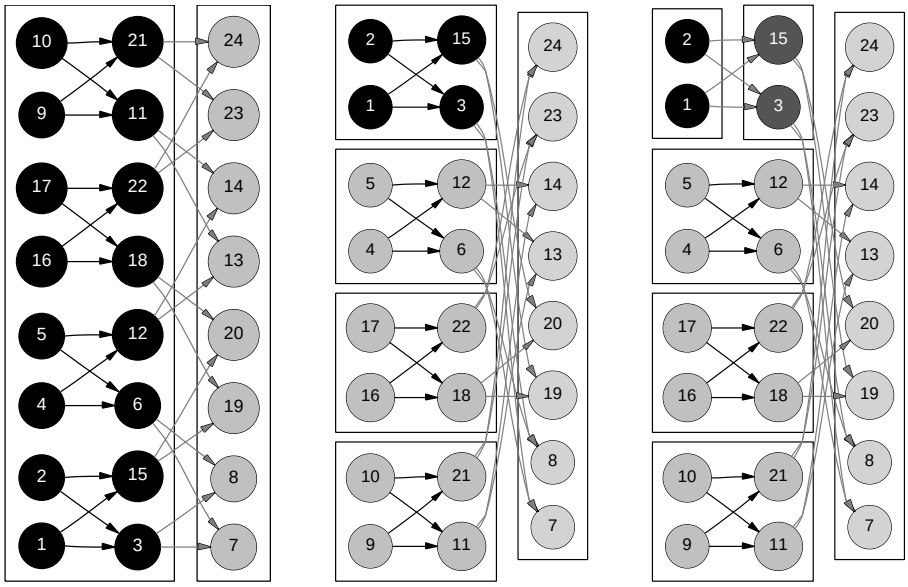

Figure 2.2: An illustration of the first three scheduling steps. (*i*) In the first scheduling step (picture on the left), the scheduler performs a serial split. The resulting two subgraphs are then scheduled separately, serializing the result. (*ii*) In the second scheduling step (picture in the middle), connected components of the upper subgraphs are separated because they are independent. (*iii*) In the third scheduling step (picture on the right), another serial split is performed in the upper-left subgraph. The shading of the nodes indicates the recursion level.

large (i. e., we will not merge two big subdags so as not to confuse the register allocator of the C compiler).

We now illustrate the first part of the scheduling phase (*sched*). We do not show the result of the second part (*asched*), because the output of asched is isomorphic to the final C output shown in Section 2.2.4.

The scheduling algorithm is a recursive procedure consisting of two steps that are alternately applied to the dag.

Serial split. This step is applied when the dag is connected. The step partitions the dag into two subdags, which are composed serially into a schedule. Informally, nodes that are closer to an input node than to an output node belong to the first subdag, otherwise they belong to the second. You can see such a serial split in the leftmost picture of Figure 2.2.

Parallel split. This step is applied when the dag consists of more than one connected component. We form a series-parallel dag by parallel composition of the schedule of each component. (Since the components are disconnected, they can be executed in any order.) The picture in the middle of Figure 2.2 shows an example of a parallel split.

2.2.4 Unparsing

This section shows the C unparsing of the assignment list of Figure 2.1. The code has been scheduled according to the partition shown in Figure 2.2.

```
void fftw_no_twiddle_4(const fftw_complex *input,
                       fftw_complex *output,
                       int istride, int ostride)
{
  fftw_real tmp3, tmp6, tmp9, tmp10, tmp11, tmp14, tmp15, tmp16;

  {
    fftw_real tmp1, tmp2, tmp7, tmp8;

    tmp1  = c_re(input[0]);                              /* node  #1 */
    tmp2  = c_re(input[2 * istride]);                    /* node  #2 */
    tmp3  = tmp1 + tmp2;                                 /* node  #3 */
    tmp11 = tmp1 - tmp2;                                 /* node #15 */
    tmp7  = c_im(input[0]);                              /* node  #9 */
    tmp8  = c_im(input[2 * istride]);                    /* node #10 */
    tmp9  = tmp7 - tmp8;                                 /* node #11 */
    tmp15 = tmp7 + tmp8;                                 /* node #21 */
  }
  {
    fftw_real tmp4,tmp5,tmp12,tmp13;

    tmp4  = c_re(input[istride]);                        /* node  #4 */
    tmp5  = c_re(input[3 * istride]);                    /* node  #5 */
    tmp6  = tmp4 + tmp5;                                 /* node  #6 */
    tmp10 = tmp4 - tmp5;                                 /* node #12 */
    tmp12 = c_im(input[istride]);                        /* node #16 */
    tmp13 = c_im(input[3 * istride]);                    /* node #17 */
    tmp14 = tmp12 - tmp13;                               /* node #18 */
    tmp16 = tmp12 + tmp13;                               /* node #22 */
  }
  c_re(output[2 * ostride]) = tmp3 - tmp6;               /* node  #7 */
  c_re(output[0])           = tmp3 + tmp6;               /* node  #8 */
  c_im(output[ostride])     = tmp9 - tmp10;              /* node #13 */
  c_im(output[3 * ostride]) = tmp10 + tmp9;              /* node #14 */
  c_re(output[3 * ostride]) = tmp11 - tmp14;             /* node #19 */
  c_re(output[ostride])     = tmp11 + tmp14;             /* node #20 */
  c_im(output[2 * ostride]) = tmp15 - tmp16;             /* node #23 */
  c_im(output[0])           = tmp15 + tmp16;             /* node #24 */
}
```

2.3 Example 2: Six Point Halfcomplex DFT

This section shows how `genfft` produces a real to complex transform of size $n = 6$.

For a (one-dimensional) transform of an array of real numbers, the property $X_i = X^*_{n-i}$ for $i = 0, 1, \ldots, n-1$ holds, where x^* denotes the conjugate complex of x. Arrays with this property are called *Hermitian*. Because of the Hermitian symmetry, only half the result of a real transform needs to be stored.

FFTW stores Hermitian arrays using the so-called *halfcomplex* array format. A halfcomplex array is an array of real numbers. A Hermitian array X is stored in a halfcomplex array Y as follows:

$$Y_i = \begin{cases} \mathrm{Re}(X_i), & i = 0, 1, \ldots, n/2, \\ \mathrm{Im}(X_{n-i}), & i = n/2+1, \ldots, n-1. \end{cases}$$

This layout is a generalization of the layout presented in [SJHB87]. The name "halfcomplex" appears in the GNU Scientific Library (GSL)[GDT+99], which uses this layout for powers-of-2 transforms. This storage scheme is useful because n_1 halfcomplex arrays, each containing a transform of size n_2, can be combined in place to produce a transform of size $n_1 n_2$, just like in the complex case. This property is not true of layouts like the one used in FFTPACK [Swa82], which stores a Hermitian array by interleaving real and imaginary parts.

The development in the rest of this section parallels that of Section 2.2.

2.3.1 The Creation Phase

Table 2.2 shows the output of the creation phase. (In the pseudo-code, we show floating-point constants with reduced precision to enhance readability. The actual constants used in the code can have arbitrary precision, and the default is 50 digits.)

The code is far from being optimal. Common subexpressions exist, such as `-Re(In[1])`. The number of multiplications can be reduced by collecting constant factors.

2.3.2 The Simplification Phase

Figure 2.3 shows the simplified fragment of pseudo-code for this example together with the corresponding data-dependency dag.

2.3.3 The Scheduling Phase

As in the previous example, we present the first steps the first part of the scheduling algorithm: Figure 2.4 shows a *serial split*, Figure 2.5 shows both a *parallel*

```
Re(Out[0]) := Re(In[0])+Re(In[3])+Re(In[2])+
              Re(In[5])+Re(In[4])+Re(In[1])
Re(Out[1]) := Re(In[0])+(-Re(In[3]))+
              (-0.5*Re(In[2])+(-Re(In[5])))+
              (-0.5*Re(In[4])+(-Re(In[1])))
Im(Out[1]) := (-0.866*Re(In[2])+(-Re(In[5])))+
              (0.866*Re(In[4])+(-Re(In[1])))
Re(Out[2]) := Re(In[0])+Re(In[3])+
              (-0.5*Re(In[2])+Re(In[5]))+
              (-0.5*Re(In[4])+Re(In[1]))
Im(Out[2]) := (-(-0.866*Re(In[2])+Re(In[5]))+
              (0.866*Re(In[4])+Re(In[1])))
Re(Out[3]) := Re(In[0])+(-Re(In[3]))+Re(In[2])+
              (-Re(In[5]))+Re(In[4])+(-Re(In[1]))
```

Table 2.2: The result of the generator phase for a 6-point real-to-halfcomplex DFT. The generator chooses the prime factor algorithm to break down the original problem of size $n = 6$ into subproblems of size $n = 2$ and $n = 3$ that are calculated by the directly applying the DFT formula (2.1).

```
 1  t1 := Re(In[0])
 2  t2 := Re(In[3])
 3  t3 := t1-t2
 4  t4 := Re(In[2])
 5  t5 := Re(In[5])
 6  t6 := t4-t5
 7  t7 := Re(In[4])
 8  t8 := Re(In[1])
 9  t9 := t7-t8
10  t10 := t6+t9
11  Re(Out[1]) := t3-(0.5*t10)
12  Re(Out[3]) := t3+t10
13  Im(Out[1]) := 0.866*(t9-t6)
14  t11 := t7+t8
15  t12 := t4+t5
16  Im(Out[2]) := -(0.866*(t11-t12))
17  t13 := t1+t2
18  t14 := t12+t11
19  Re(Out[2]) := t13-(0.5*t14)
20  Re(Out[0]) := t13+t14
```

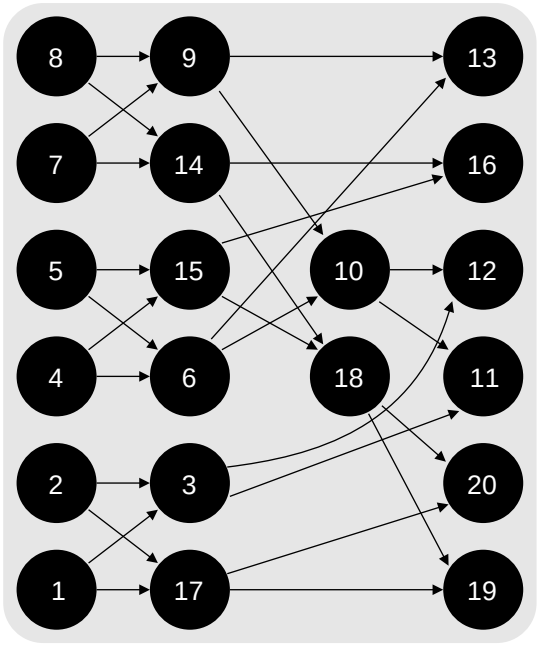

Figure 2.3: The result of the simplifier phase for Example 2. The left-hand side again shows an assignment list. As in the previous example, temporary variables have been introduced (for loading inputs and for holding common subexpressions). Also notice that all constants have been made positive: This step helps to store pairs of constants $(x, -x)$ more efficiently resulting in better performance of the code produced. The right-hand side presents the output of the simplifier phase in the form of a dag. Again, the node numbers refer to line numbers on the left-hand side of the figure. Edges in the graph indicate data dependencies.

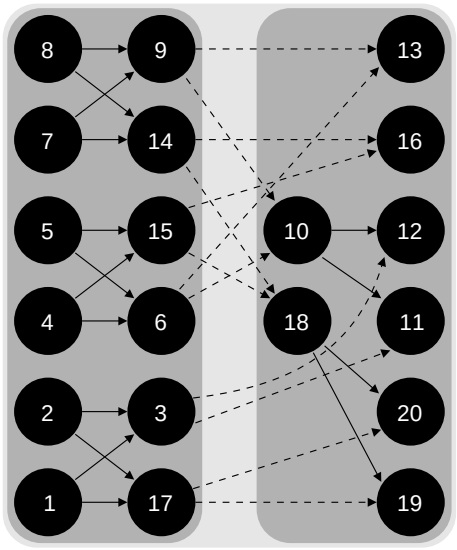

Figure 2.4: First scheduling step. Because there is only one connected component in the original dag (Figure 2.3), a serial split is performed. Both subgraphs are scheduled separately, serializing the results.

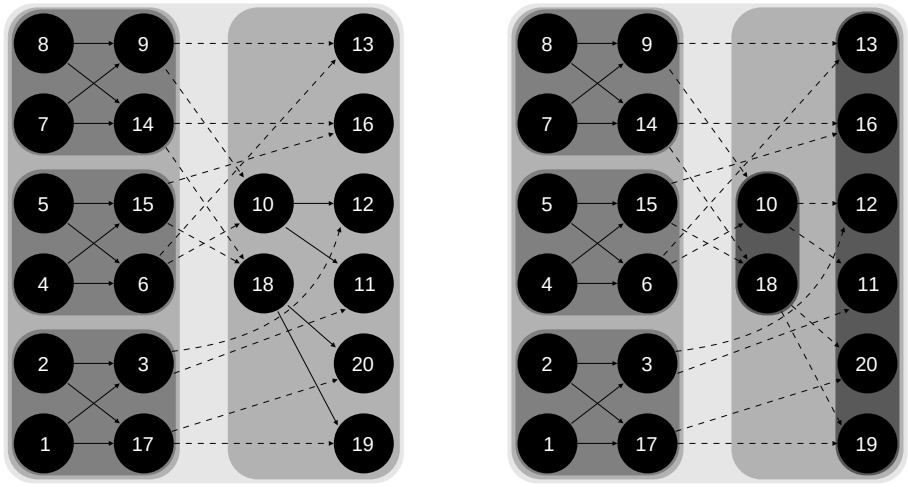

Figure 2.5: Second and third scheduling step. The illustration of the second scheduling step (on the left-hand side) shows a partitioning of the subgraphs into their connected components (parallel split of the subgraphs). Because connected components within a subgraph are independent, the second part of the scheduling phase (*asched*) can reorder them aiming at better machine utilization. The third step shows another case of serial split.

and another *serial split*.

Figure 2.5 and Figure 2.2 employ different typographic conventions. In the previous example, we wanted to emphasize how the scheduling algorithm traverses the data-dependency dag (the input of the algorithm). Here, the emphasis is on the output of the algorithm, i. e., on the series-parallel structure of the schedule.

2.3.4 Unparsing

As in the previous example, we now present final C code generated by `genfft`.

The generator supports different styles of the declaration/definition of constants (either in a "pre-processor style" (via `#define`, or in a "static-const style"). In this example we show the latter style.

```
static const fftw_real K500000000 = FFTW_KONST(+0.50000000);
static const fftw_real K866025403 = FFTW_KONST(+0.86602540);

void fftw_real2hc_6(const fftw_real *input,
                    fftw_real *real_output, fftw_real *imag_output,
                    int istride, int real_ostride, int imag_ostride)
{
  fftw_real tmp1,tmp2,tmp3,tmp6,tmp9,tmp10,tmp11,tmp12,tmp13,tmp14;

  tmp1  = input[0];                                        /* node  #1 */
  tmp2  = input[3 * istride];                              /* node  #2 */
  tmp3  = tmp1 - tmp2;                                     /* node  #3 */
  tmp13 = tmp1 + tmp2;                                     /* node #17 */
  {
    fftw_real tmp4, tmp5, tmp7, tmp8;

    tmp7  = input[4 * istride];                            /* node  #7 */
    tmp8  = input[istride];                                /* node  #8 */
    tmp9  = tmp7 - tmp8;                                   /* node  #9 */
    tmp11 = tmp7 + tmp8;                                   /* node #14 */
    tmp4  = input[2 * istride];                            /* node  #4 */
    tmp5  = input[5 * istride];                            /* node  #5 */
    tmp6  = tmp4 - tmp5;                                   /* node  #6 */
    tmp12 = tmp4 + tmp5;                                   /* node #15 */
  }

  imag_output[imag_ostride] =
          K866025403 * (tmp9 - tmp6);                      /* node #13 */
  tmp10 = tmp6 + tmp9;                                     /* node #10 */
  real_output[real_ostride] =
          tmp3 - K500000000 * tmp10;                       /* node #11 */
  real_output[3 * real_ostride] = tmp3 + tmp10;            /* node #12 */
  imag_output[2 * imag_ostride] =
          -(K866025403 * (tmp11 - tmp12));                 /* node #16 */
  tmp14 = tmp12 + tmp11;                                   /* node #18 */
  real_output[2 * real_ostride] =
          tmp13 - K500000000 * tmp14;                      /* node #19 */
  real_output[0] = tmp13 + tmp14;                          /* node #20 */
}
```

Chapter 3

Core Modules

This chapter describes the following core modules: `Genfft` (Section 3.1), `Fft` (Section 3.2), `Exprdag` (Section 3.3), `Schedule` (Section 3.4), `Asched` (Section 3.5), `To_c` (Section 3.6). All sections in this chapter share the following common structure.

Description briefly explains the purpose of the module.

Data Types summarizes the major data types declared and used in the module.

Main Functions describes major functions implemented in the module.

3.1 Genfft: Program Entry Point

Description

This module contains the entry point for the `genfft` program. The module parses the command-line parameters, initializes the magic variables that control global parameters, and calls the rest of the program as needed to generate the code requested by the user.

Data Types

```
type codelet_type =
  | TWIDDLE
  | NO_TWIDDLE
  | REAL2HC
  | HC2HC
  | HC2REAL
  | REALEVEN
  | REALODD
  | REALEVEN2
  | REALODD2
  | REALEVEN_TWIDDLE
  | REALODD_TWIDDLE
```

Table 3.1: The data type `codelet_type`.

`codelet_type` is an enumeration type of all supported codelet types.

`mode` is an enumeration type of all supported codelet types, plus a number indicating the transform length of the DFT. An instance of this type is used during argument parsing in the function `main`.

Main Functions

Function name	*Function type*
`list_to_c`	`int list -> string`
`make_expr`	`string -> Expr.expr`
`optimize`	`Exprdag.dag -> Asched.annotated_schedule`
`codelet_description`	`int -> Fft.direction -> codelet_type -> To_c.c_fcn -> string`
`fftw_no_twiddle_gen`	`int -> Fft.direction -> string`
`athena_no_twiddle_gen`	`int -> Fft.direction -> string`
`no_twiddle_gen`	`int -> Fft.direction -> string`
`fftw_twiddle_gen`	`int -> Fft.direction -> string`
`athena_twiddle_gen`	`int -> Fft.direction -> string`
`twiddle_gen`	`int -> Fft.direction -> string`
`main`	`unit -> 'a`

Table 3.2: Function types of functions defined in `Genfft`.

Functions Named ..._..._gen

Many of the functions in this module have the form `..._..._gen n dir`, e.g., `fftw_no_twiddle_gen`. All of these functions use a specific function (in module `Fft`) to compute an intermediate representation of the DFT computation (a dag). These functions first optimize the dag (see Section 3.3), then they schedule it, and finally they invoke the unparser to produce C code.

"Unparsing" is the transition from abstract code to actual program text. Unparsing support for variables helps to hide irrelevant information about variables. See Section 4.9 in Chapter 4 for more information about the unparsing of variables.

All functions prefixed `athena_...` are experimental routines that are not documented in the following text.

`no_twiddle_gen n dir` tests a global parameter (`Magic.athenafft`) and either calls `fftw_no_twiddle_gen` or `athena_no_twiddle_gen` to perform the computation requested.

`fftw_no_twiddle_gen n dir` uses the function `Fft.no_twiddle_gen_expr` (see Section 3.2) to generate the DFT computation for the given transform length n and direction `dir`. It uses a variable unparser for input and output arrays, each with different strides. There is no unparsing support for twiddle factors.

`twiddle_gen n dir` tests a global parameter (`Magic.athenafft`) and either calls `fftw_twiddle_gen` or `athena_twiddle_gen`.

`fftw_twiddle_gen n dir` uses the function `Fft.twiddle_dit_gen_expr` to generate the FFT computation. The input and output arrays are aliased. There is a stride for this array. Twiddle factors are used, but they have no stride. A loop is generated to iterate over the data, using an offset (per round) of `dist`. The offset of the twiddle factor data is calculated by `Twiddle.twiddle_policy`. The number of iterations is determined by an argument of the function generated, `m`.

Top-level Functions

The function `make_expr` takes the name of a variable (as a string), converts it into an expression and returns it.

`list_to_c` maps a list of integers to a string (C code), e. g., `(list_to_c [1;2;3])` returns the string `"1, 2, 3"`. It is used to print out the order of the twiddle factors used in a codelet.

`optimize` takes an intermediate representation of the DFT code, calls the simplifier to apply various optimizations (algebraic expression simplification, common subexpression elimination, scheduling and reordering) and returns a form suitable for unparsing.

`codelet_description` creates a C structure that includes information about a given codelet (DFT type, transform length and direction, information about twiddle factors). That C structure is used to integrate a codelet into the FFTW system.

`main` parses all command line options, sets all global parameters, chooses the right codelet generator, calls it to create the requested code and displays the code on `stdout`. In case of an error, an appropriate error message is displayed (usually "too many arguments" or the `usage` string).

3.2 Fft: Generation of the FFT Computation

Description

This module contains the generator of `genfft`. It consists of a number of codelet generators for specific DFT types, directions and transform lengths.

Data Types

```
type direction =
  | FORWARD
  | BACKWARD
```

Table 3.3: The data type `direction`.

The only data-type declared in this module is `direction` denoting the direction of the DFT.

Main Functions

Function name	*Function type*
`@*`	`Complex.expr -> Complex.expr -> Complex.expr`
`@+`	`Complex.expr -> Complex.expr -> Complex.expr`
`@-`	`Complex.expr -> Complex.expr -> Complex.expr`
`choose_factor`	`int -> int`
`freeze`	`int -> (int -> 'a) -> int -> 'a`
`sign_of_dir`	`direction -> int`
`conj_of_dir`	`direction -> Complex.expr -> Complex.expr`
`dagify`	`int -> Symmetry.symmetry ->` `(int -> Complex.expr) -> Exprdag.dag`
`fftgen_prime`	`int -> (int -> Complex.expr) ->` `int -> int -> Complex.expr`
`fftgen_rader`	`int -> (int -> Complex.expr) ->` `int -> int -> Complex.expr`

`fftgen`	`int -> Symmetry.symmetry ->`
	`(int -> Complex.expr) ->`
	`int -> int -> Complex.expr`

`no_twiddle_gen_expr`	`int -> Symmetry.symmetry -> direction ->` `Exprdag.dag`

`twiddle_dit_gen_expr`	`int ->`
	`Symmetry.symmetry -> Symmetry.symmetry ->`
	`direction -> Exprdag.dag`

`twiddle_dif_gen_expr`	`int ->`
	`Symmetry.symmetry -> Symmetry.symmetry ->`
	`direction -> Exprdag.dag`

Table 3.4: Function types of all functions defined in `Fft`.

Auxiliary Functions

The functions `(@*)`, `(@+)` and `(@-)` directly map to the functions `Complex.times`, `Complex.plus` and `Complex.uminus`. Because they can be used as (binary) infix operators, their use enhances the readability of the program text.

`choose_factor n` calculates and returns a suitable factor of n. A suitable factor is either i such that $gcd(i, n/i) = 1$ with i as big as possible or else the biggest factor i of n, where $i < \sqrt{n}$. This function is used in various DFT generation algorithms that depend on partitioning a DFT of initial size n into smaller DFTs. (e. g., `prime_factor` and `cooley_tukey`). Whenever possible, the prime factor algorithm is used instead of the Cooley Tukey algorithm.

`freeze` implements lazy arrays. `freeze n f` returns a function `g`. When applying `g` to some integer i (with $0 \leq i < n$), freeze evaluates `(f i)`, caches the result, and returns it.

`sign_of_dir dir` maps a direction `dir` to an integer (`-1` for `FORWARD`, `1` for `BACKWARD`).

`conj_of_dir dir` returns a function to calculate the conjugate complex of a value, depending on the direction of the DFT. For forward transforms it returns ϵ, `conj` (see Section 4.1 otherwise. The function is used for adapting twiddle factors depending on the transform direction.

`dagify n sym f` takes a frozen DFT-computation of size n and maps it to an expression dag (of type `Exprdag.dag`) exploiting its output data symmetries.

Core of the DFT Generator

`fftgen n sym input dir` tries to factor n into some n_1 and n_2, decides what DFT algorithm is most appropriate and calls it, returning a frozen expression of the DFT computation. For an explanation the implementation of the Cooley-Tukey DFT algorithm, see [Fri99] page 5.

`fftgen_prime n input sign` returns an expression denoting the calculation of an DFT of size n. The function `input` is used to obtain input variables (with exploiting input symmetries). `sign` is either -1 (forward) or 1 (backward). If n is greater or equal `Magic.rader_min`, another function (`fftgen_rader`) is used for calculating the DFT. Otherwise the result of a direct application of DFT-equation is returned. If n is odd, parts of the expression are reordered.

Functions named ..._..._gen_expr

`no_twiddle_gen_expr n sym dir` maps the direction `dir` to an integer, calls `fftgen` to generate the DFT expression without using any input symmetries (see Section 4.6), and maps the result to an expression dag.

`twiddle_dit_gen_expr n sym1 sym2 dir` generates the code for a DFT of size n. It performs the following steps:

- `sym1` is used as the symmetry of the input values.
- Input values are scaled by twiddle factors. Depending on the direction, twiddle factors are adapted (by using `conj_of_dir`). The current twiddle-loading policy (see Section 4.7 for information about several policies available) is used.
- `fftgen` is called to calculate the DFT expression using the symmetry `sym2`.
- `dagify` is used to map the result to an expression dag using `sym2` as an output symmetry.

`twiddle_dif_gen_expr n sym1 sym2 dir` is similar to the function `twiddle_dit_gen_expr`, but:

- `sym1` is not used as a symmetry of the inputs, but as an intermediate symmetry.
- `fftgen` is used to calculate the DFT expression using the symmetry `sym1`.
- The intermediate result is scaled by twiddle factors (as before).
- The result is converted to an expression dag using `sym2` as an output symmetry.

Within the DFT generation phase, it is convenient to work with complex-expressions. As a support for programming languages do not offer support for handling complex values (e.g., C), a transition from complex-to-real takes place at the end of this phase (this is done by `dagify` when it produces an the output expression and stores the single values in the appropriate place). All following phases (the simplification phase, the scheduler and so on) operate on real-arithmetic expressions.

3.3 Exprdag: Expression Simplification

Description

`Exprdag` is the most complex module of `genfft`. This module simplifies the arithmetic complexity of an expression dag. The `Exprdag` module contains 10 auxiliary submodules: `Hash`, `LittleSimplifier`, `AssocTable`, `StateMonad`, `MemoMonad`, `Oracle`, `Reverse`, `Stats`, `AlgSimp`, and `Destructor`.[1]

[1] These modules will be split in the next FFTW release.

Data Types

An expression dag is represented by the following `node` data type.

```
type node =
  | Num of Number.number
  | Load of Variable.variable
  | Store of Variable.variable * node
  | Plus of node list
  | Times of node * node
  | Uminus of node
```

Table 3.5: The data type `node`.

An expression is either a number, a load of an input variable or twiddle factor, a store of a node into an output variable, the sum of a list of nodes, the product of two nodes, or the negation $-x$ of a node x.

The data type `node` is not a tree, and subnodes are intended to be shared. The code is written as if `node` were a tree, however, because it is easier to write it in this way. The fact that nodes are shared is encapsulated within monads. (See Section 3.3.5.)

Main Functions

The actual code of the module is contained in the 10 submodules. We now turn our attention to these submodules.

3.3.1 Hash: Hashing of Nodes

This module exports three functions: `hash_float`, `hash_variable`, and `hash_node`. Each function associates an integer to a value, where a "value" is a number, a variable, or a dag node, respectively.

Associative tables (see Section 3.3.3) use hash values as keys.

3.3.2 LittleSimplifier: Quick Simplifier

This module implements a small set of local algebraic simplifications that can be executed quickly. Each function produces a `node`, removing trivial operations without being concerned with common subexpression elimination and other global transformations.

This module is redundant, but it improves the speed of the system by passing a smaller dag to the expensive optimizer `AlgSimp`.

`makeNum` creates a `Num` node. It doesn't do anything.

`makeUminus` create a `Uminus` node, simplifying $-(-x)$ into x.

`makeTimes` create a `Times` node. This function simplifies $1 \cdot x$ to x, $0 \cdot x$ to 0, and it multiplies two constants instead of creating a `Times` node representing the product.

Similarly, `makePlus` produces a `Plus` node, while simplifying additions. The function first reduces the sum, i.e., it adds together all constants. Then it performs these simplifications: $a + a \mapsto 2a$, $ab + ad \mapsto a(b + d)$.

3.3.3 AssocTable: Functional Associative Tables

This module implements associative tables, i.e., maps from keys to values. Our associative tables are purely functional: once created, they never change. Updating a table produces a new table.

Associative tables are implemented by means of (unbalanced) binary search trees indexed by integers. Since tables are intended to store dag nodes, `genfft` uses hash functions to map nodes into integers (non uniquely).

The constant `empty` is an empty hash table.

The function `lookup hash equal key table` returns `table`(`key`), if any, or `None` if no such value exists in the table. `lookup` employs a hash function for mapping keys into integers, and the `equal` function to compare keys.

The function `insert hash key value table` returns a new table that extends `table` with the association (`key`, `value`). If `key` belongs to `table`, then the effect is undefined.

The functions `node_insert` and `node_lookup` specialize tables to dag nodes, using the hash function from Section 3.3.1 and the equality predicate `==`.

3.3.4 StateMonad: State Monad

This module implements the *state monad* from [Wad92, Section 2.5].

Monads have a beautiful theory and elegant applications [Wad97]. For the purposes of these reports, however, we can treat monads as just another modularization technique in the same class as function calls, "objects" as used in object-oriented programming, and modules. In `genfft`, the main use of monads is in module `AlgSimp` (see Section 3.3.9), where they perform two functions: they allow the simplifier to treat the expression dag as if it were a tree, making the implementation considerably easier, and they perform common-subexpression elimination.

Programs that use monads are called monadic-style programs. For our purposes, in a monadic-style program you convert all expressions of the form

```
let x = a in (b x)
```

into

```
a >>= fun x -> returnM (b x)
```

This code should be read "call `f`, and then name the result `x` and return `(b x)`." The advantage of this transformation is that the meanings of "then" (the infix operator `>>=`) and "return" (the function `returnM`) can be defined so that they perform all sorts of interesting activities, such as carrying state around, perform I/O, act nondeterministically, etc. (See [Wad92].)

The monad defined in the `StateMonad` module defines the two operators `>>=` and `returnM`. The monad implicitly propagates a state, which is called `s` in the code. You can access the state with the function `fetchState`, and you can modify the state with the function `storeState`. For example, the following monadic program fetches the state (an integer), increments it, stores the incremented value into the state, and returns the new state.

```
let program () =
  fetchState >>= fun s ->
    storeState (s + 1) >>= fun () ->
      fetchState >>= fun s ->
        returnM s
```

The monadic program is not directly executable, i.e., if you invoke `program ()` nothing happens. The function `runM program arg initial_state` runs the program.

```
# runM program () 1;;
- : int = 2
```

For convenience, the `StateMonad` module defines two auxiliary functions, namely, `>>` and `mapM`. The function `>>` is used instead of `>>=` when we do not care about the value that `>>=` passes to the function to its right. Using `>>`, you can write `a >> b` instead of the longer `a >>= fun _ -> b`. If `l` is a list, the function `mapM f l` applies `f` element-wise to the elements of `l` and returns a list of the results, in order. `mapM` is thus the analogous of `map` for monadic-style programs. Unline `map`, `mapM` traverses `l` from left to right because the direction of state propagation is important.

You may wonder why `genfft` uses such a machinery just to keep a state variable. Indeed, earlier versions of `Exprdag` used the imperative features of Objective Caml for this purpose. I (Matteo Frigo) switched to a monadic-style

implementation for two reasons. First, I wanted to experiment with nondeterministic simplifiers that would try several mutually-exclusive simplifications. (The experiment did not lead to any improvement, and it does not survive in the distributed code.) Second, bugs were hard to find in the imperative implementation. For example, a table was not properly initialized between consecutive invocations of the simplifier, leading to incorrect results. This problem cannot happen in the monadic-style program because the initial state must be specified explicitly. I now believe that the syntactic noise introduced by the monadic combinators is a fair price to pay for less error-prone programs.

3.3.5 MemoMonad: Monad with Memoization

This module use the state monad (see Section 3.3.4) to memoize a function. *Memoization* consists in storing the result of a function evaluation into a table so that the result can be used later without having to evaluate the function again. Memoization is used by module `AlgSimp` (see Section 3.3.9) to avoid optimizing the same node twice.

The function `memoizing` is written in monadic style using the state monad.

3.3.6 Oracle: Decisions Concerning the Canonical Form

The oracle module exports a single function

```
val should_flip_sign : node -> bool
```

The function `should_flip_sign` determines whether the canonical form of the expression $a - b$ is $a - b$ or $b - a$. Canonical forms reduce the number of cases to be examined during common-subexpression elimination.

Obtaining the right canonical form is tricky. We do not want to change the sign of expressions that only appear once, because we may introduce extra work. On the other hand, we want to recognize expressions that appear more than once in the dag.

The oracle works as follows. It evaluates a node at a random point of the input space, that is, it fixes the input variables to an arbitrary random value and it evaluates the node numerically. The oracle holds a table of all previously encountered numerical evaluations, and it considers two keys to be equal if their numerical value is approximately the same in absolute value. The sign of a node should be changed if the sign of the node evaluation differs from the sign of the value in the table. This case denotes either that the same expression already appeared with the opposite sign in the expression tree, or that two unrelated nodes collide into the same evaluation. Collisions are extremely rare by design. Even if a collision occurs, the effect is to flip some sign unnecessarily without affecting the correctness of `genfft`.

3.3.7 Reverse: Dag Transposition

This module implements the dag transpositiong algorithm described in [Fri99].

3.3.8 Stats: Dag Statistics

This module computes the arithmetic complexity of a dag, i.e., the number of additions, multiplications, sign negations, floating point constants, and load and store operations in a dag. The function `complexity` computes the complexity of a dag. This module is only used for debugging.

3.3.9 AlgSimp: Algebraic Simplification

This module is the core of the `genfft` simplifier. This module performs algebraic simplification and common-subexpression elimination. The code is written in monadic style (see Section 3.3.4).

The simplifier is written as if it were simplifying a tree instead of a dag. Behind the scenes, however, the monadic operators perform two activities. First, the whole simplifier is memoized using the module `MemoMonad`. Consequently, the simplifier never traverses the same dag node twice, making the dag look like a tree from the point of view of the simplifier. Second, whenever the simplifier produces a new simplified node, a monadic operator checks whether an equivalent node was already produced by the simplifier in the past. In this case, the new node is discarded and the simplifier returns the old node. By a suitable definition of "equivalent node", this process accomplishes common-subexpression elimination (CSE).

The monadic state of the simplifier consists of two components, one for the memoization of already visited nodes, and the other for CSE.

The first four functions of the module, namely, `fetchSimp`, `storeSimp`, `lookupSimpM`, and `insertSimpM`, implement the transformation of the dag into a tree. `fetchSimp` and `storeSimp` fetch and store, respectively, the proper component of the state. The other two functions look up a node in and, respectively, insert a node into the table of already simplified nodes.

The next group of functions, namely, `equalCSE`, `fetchCSE`, `storeCSE`, `lookupCSEM`, `insertCSEM`, `identityM`, and `makeNode`, implement common-subexpression elimination. CSE is implemented by memoization of the identity function. Recall that memoization uses the associative table operations, and that the table operations employ an equality predicate for retrieving data stored into the table. (See Section 3.3.3.) CSE employs the special equality predicate `equalCSE`, which considers expressions such as $a \cdot b$ and $b \cdot a$ to be the same. The simplifier calls `makeNode` whenever it produces a new simplified dag node.

The following functions implement the simplifier proper. The main entry point is the function `algsimpM`, which works bottom-up on the expression tree (recall

that the dag looks like a tree because of memoization). `algsimpM` first recursively calls itself to simplify subnodes of an expression, if any, and then dispatches one of the ancillary functions `snumM`, `splusM`, `stimesM`, and `suminusM`, depending on the node type.

`snumM` transforms a negative number a into `Uminus` $-a$. This transformation makes all constants positive.

`suminusM` transforms $-(-a)$ into a and -0 into 0.

`stimesM` transforms $(-a)b$ into $-(ab)$, $a(-b)$ into $-(ab)$, $1 \cdot a$ into a, $0 \cdot a$ into 0, $(-1) \cdot a$ into $-a$. Moreover, `stimesM` replaces the product of constants by the constant product.

The remaining ancillary function `splusM` is more complicated. `splusM` first calls `mangleSumM` to obtain a first simplification of a sum, and then it canonicalizes the output. In turn, `mangleSumM` uses other functions to perform more transformations. Specifically, `reduce_sumM` computes the sum of all constants and removes trivial additions of 0. `collectExprM` transforms $ja + ka$ into $(j + k)a$, where j and k are constants. `collectCoeffM` transforms $ja + jb$ into $j(a + b)$, where j is a constant. The function `eliminateButterflyishPatternsM` simplifies expressions of the form $(a + b) \pm (a - b)$. These expressions occur when a variant of Rader's algorithm [Rad68] is used to generate transforms of size 13, and in other cases.

Finally, the monadic simplifier `algsimpM` is run by the nonmonadic function `algsimp`, which runs the monadic code.

Right after the module `AlgSimp` in the code, we find another function called `algsimp`. This function alternatively simplifies (using `AlgSimp.algsimp`) the dag and its transposed form. The rationale behind this algorithm is described in [Fri99].

3.3.10 Destructor: Destruction of a Dag into an Assignment List

This module exports the function `to_assignments` that converts a dag into a list of assignments. (See Section 4.3 for the definition of the type `assignment`.)

In an assignment list, most values have a temporary name. For example, the expression dag `Store (v, Plus [Times (a, b); c])` may be converted into `[Assign (t1, Times(a, b)); Assign (v, Plus [t1; c])]`. This module thus performs two functions: it determines which values should be assigned to a named temporary variable, and it rewrites dag nodes to use the new temporaries.

The code is written in monadic style, and it traverses the dag twice. (See function `peek_alistM` at the end of the module.)

In the first pass (`visit_rootsM`), `genfft` traverses the dag and counts how many times a node is referenced by other nodes. This bookkeping is accomplished by the function `counting`.

The second pass (function `expr_of_nodeM`) decides which nodes should be assigned to a temporary variable and which should be inlined. The inlining policy is expressed in terms of three auxiliary functions `inlineM`, `with_tempM`, and `with_temp_maybeM`. `inlineM` does not create a temporary variable, and it expands the expression inline. In the released code, floating-point constants are never assigned to temporary variables. The `with_tempM` policy creates a temporary variable. In the code, `Load` expressions are always assigned to temporary variables (unless the flag `Magic.inline_loads` is set). `with_tempM` creates a fresh temporary variable, creates an assignment of the node to the variable, stores the assignment into the monadic state, and returns an expression that references the temporary variable. The `with_tempM` policy assigns a node to a temporary if the node is referenced more than once, as determined by the first pass.

The effect of the various inlining policies on the speed of the resulting code has never been studied extensively, and this is a topic for further experimentation. The current choice produces code similar to what I (Matteo Frigo) find most readable.

The module also contains undocumented experimental code for producing fused multiply-add instructions (FMA), as found, for example, in PowerPC and MIPS processors.

3.4 Schedule: Efficient Ordering of Instructions

This phase analyses the structure of a dag with regard to data dependencies. It captures the structure in a dag that explicitly shows dependent and independent parts of the computation.

Description

This transformation and optimization phase immediately follows the expression simplification phase implemented in module `Exprdag` (see Section 3.3).

This phase maps a list of assignments produced by the preceding phase to a series-parallel dag.

The output of this phase (a value of type `schedule`) forms the basis for the next step, i. e., the process of reordering and serializing independent code blocks to maximize performance (see Section 3.5).

Data Types

At the end of the previous step a computation rule for a specific FFT kernel was represented as a list of assignments, i. e., in a linear fashion. This step tries to find out more about the internal structure of the computation, so a recursive data type (a tree) is used:

```
type schedule =
  | Done
  | Instr of Expr.assignment
  | Seq of (schedule * schedule)
  | Par of schedule list
```

Table 3.6: The data type `schedule`. There are four different types of schedules: `Done` represents the empty schedule. `Instr(a)` represents a schedule consisting of exactly one assignment. `Seq(s0,s1)` represents an ordered sequence of two schedules, `s0` and `s1`. `s1` depends on `s0` and can therefore only be executed after `s0` has finished. `Par(ps)` represents an unordered set of an arbitrary number of independent sub-schedules. Independent means that all schedules in `ps` can be executed (*i*) in parallel and/or (*ii*) in arbitrary order. All arrangements will lead to the same (correct) result. The goal of the next step (in module `Asched`) is to find a favorable ordering for the purely sequential case that is beneficial with regard to cache-performance.

Main Functions

Function name	*Function type*
to_assignment	Dag.dagnode -> Expr.assignment
makedag	Expr.assignment list -> Dag.dag
return	'a -> 'a
has_color	Dag.color -> Dag.dagnode -> bool
set_color	Dag.color -> Dag.dagnode -> unit
has_either_color	Dag.color -> Dag.color -> Dag.dagnode -> bool
cc	Dag.dag -> Dag.dagnode list -> Expr.assignment list * Expr.assignment list
connected_components	Expr.assignment list -> Expr.assignment list list
loads_twiddle	Dag.dagnode -> bool
partition	Expr.assignment list -> Expr.assignment list * Expr.assignment list
schedule	Expr.assignment list -> schedule

Table 3.7: Function types of all functions defined in `Schedule`.

The only function that is exported via the interface file is `schedule`. The functions `schedule_alist` and `schedule_connected` are not present in Table 3.7 because they are private to the function `schedule`. Both have the same function type as `schedule`: `Expr.assignment list -> schedule`.

Handling of DAGs

In many of the scheduling functions dags (directed acyclic graphs) are used to represent the dependencies between assignments. A dag is made up of dag nodes.

The most important parts of a single node are the assignment it corresponds to, the list of predecessor nodes, the list of successor nodes, and the node's color. Section 4.2 provides a more detailed description of the structure of a dag and of various common operations on dags. The following functions are used for handling dags conveniently:

`to_assignment n` maps a dag-node `n` to an assignment, i.e., it simply strips off all of the dag specific information.

`makedag alist` maps an assignment list `alist` to a dag that reflects the dependencies between the assignments.

`return` is the identity function.

`has_color c n` returns `true` if the given node `n` has the color specified, `c`.

`set_color c n` destructively updates the color attribute of the given node `n`, it is set to `c`.

`has_either_color c1 c2 n` is `true` if the given dag-node `n` has one of two given colors, `c1` and `c2`.

Connected Components in the DAG

The second group of functions handles connected components of a dag. Different connected components represent independent parts of the FFT computation.

`cc dag inputs` maps a dag to a pair of type `Expr.assignment list * Expr.assignment list`. The first component of this pair is a list of nodes in `dag` that are reachable from the *first* input-node (`hd inputs`). The second component is a list of nodes in `dag` that are *not* reachable. If no input node (a node without a predecessor) exists in `dag`, a `failure` exception (`connected`) is raised. Every node is reachable from *some* input node per definitionem (otherwise it would be an input node itself).

`connected_components alist` maps a list of assignments `alist` to a list of assignment lists, i.e., it partitions a list of assignments. The various elements of the list returned are the connected components of the graph. Its implementation works as follows. First all input nodes (nodes without a predecessor) are collected. All nodes reachable (both predecessors *and* successors) from the first input node are removed from the dag. These nodes form the first connected component of the graph. The same method is used recursively to deliver all connected components of the remaining graph. As soon as the entire graph is partitioned, the computation terminates.

Graph Partitioning and Scheduling

The main task of the functions in this module is to map a list of assignments to a schedule (a tree representing the structure of the computation).

`loads_twiddle node` returns `true` if the dag-node `node` is an input node that represents a twiddle factor (nodes of this kind are also called "special input nodes"). Nodes of this type have exactly one input node and no predecessor.

`partition alist` generates a dag corresponding to the assignment list `alist`, colors the graph according to various data-dependency rules, splits it up into two parts (nodes "attracted" to the input side and nodes "attracted" to the output side of the graph) and returns the two parts after mapping them to assignment lists again. The function works as follows. First `alist` is mapped to a equivalent dag. Initially the color of all nodes is set to black (unused). All input nodes (nodes without a predecessor) are colored red, all special input nodes (inputs that read a twiddle factor) become yellow and all output nodes (nodes without a successor) are colored blue. Two rules are used for coloring the graph: `loopi` looks for all black (unassigned) nodes whose predecessors are all either red or yellow (normal or special inputs). It colors these nodes and their predecessors red (normal input). `loopo` looks for all black or yellow (unassigned or special input) nodes whose all successors are blue (output). All of these nodes are colored blue. These two rules are used alternately until a fixpoint is reached. A parameter (`Magic.loopo`) determines the starting rule. Finally, an integrity check is performed: If there are no red nodes, all input nodes (whether special or not) are colored red again (the partition is corrected).[2]

`schedule_alist ls` maps a list of assignments to a schedule. If `ls` has a length of 0 or 1, the mapping is obvious: An empty list is mapped to `Done`. A list with one element `a` is mapped to `Instr(a)`. Otherwise, the list (and the corresponding graph) is split up into its connected components (`connected_components`, `Par`-case). If the entire graph is connected, the scheduling is calculated by `schedule_connected` (`Seq`-case). If there are at least two connected components in the graph, the independent components are scheduled separately by a recursive call to `schedule_alist`.

`schedule_connected alist` schedules one connected graph component. It splits the graph component into two halves (with `partition`), scheduling both halves and ordering the results.

3.5 Asched: Generating Annotated Schedules

The goal of this step is to serialize independent parts of a schedule in a way that optimizes register usage by minimizing the number of register spills.

[2]The main purposes of this is to guarantee termination of the scheduling process.

Description

The previous step maps a list of assignments to a schedule. Unlike an expression list, a schedule contains not only instructions but also information about dependencies between code blocks.

What remains to be done (to get C code) is (*i*) to sequentialize the independent (`Par`) code blocks that are present in a schedule and (*ii*) to find a good nesting of the code blocks in order to convey as much information concerning temporary-variable life-spans to the C compiler, as well as unparsing itself. Both techniques (finding an efficient ordering of independent code blocks and finding appropriate levels of nesting for variable declaration) can help the register allocator of the C compiler to generate more efficient code and are implemented in this module, `Asched`.

The output of this phase—a so called "annotated schedule"—is directly fed into the unparser which generates the actual C code.

Data Types

Just like the central data type of the previous phase, the central data type of this one is a tree structure: Obviously, it *is* `schedule` minus the `Par` case (independent code blocks are sequentialized in this phase).

```
type annotated_schedule =
  Annotate of variable list * variable list * variable list *
              int * aschedule
and aschedule =
  | ADone
  | AInstr of assignment
  | ASeq of (annotated_schedule * annotated_schedule)
```

Table 3.8: The data types `aschedule` and `annotated_schedule`.

In the following the term "annotated code block" will refer to an instance of `annotated_schedule`. We will refer to the components of an instance of `annotated_schedule` as *vl1*, *vl2*, *vl3*, *d* and *asch*. These components have the following meaning:

vl1 All variables contained in this list are alive at the beginning of the annotated code block.

vl2 All variables in this list are temporary variables that are used in this block and that have a life-span reaching beyond the scope of this block. They have to be declared in some parent block of this one.

vl3 Temporary variables that have a life span limited to this block can be declared locally. Variables of this kind are in this list.

d This integer value is the sum of the number of variables declared in this block (`length vl3`) and the biggest number of declared variables in any subblock. It is the maximal number of temporary variables that are declared (and alive) at any point.

Main Functions

The only function exported via the interface file is `annotate`. It is the mapping of a schedule to an annotated schedule. The functions `analyse` and `really_analyse` are missing in Table 3.9 because they are internal to the function `annotate`. They both have the type `Variable.variable list -> Schedule.schedule -> annotated_schedule`.

Function name	*Function type*
addelem	'a -> 'a list -> 'a list
union	'a list -> 'a list -> 'a list
diff	'a list -> 'a list -> 'a list
minimize	('a -> int) -> 'a list -> 'a
uniq	'a list -> 'a list
find_block_vars	Schedule.schedule -> Variable.variable list
has_similar	Variable.variable -> Variable.variable list -> bool
overlap	Variable.variable list -> Variable.variable list -> int
reorder	Schedule.schedule list -> Schedule.schedule list
rewrite_declarations	bool -> annotated_schedule -> annotated_schedule
annotate	Schedule.schedule -> annotated_schedule

Table 3.9: Function types of all functions defined in `Asched`.

Generic List Functions

In this module — like in the previous one, `Schedule` — there is a set of generic functions that might as well be in another module, say `Util`. As an alternative, standard library functions might have been used more extensively.

Because they operate on plain (i. e., unordered) lists, one-operand functions (`addelem`, `minimize`) have linear complexity; two-operand functions (`union`, `diff`) and `uniq` have quadratic complexity.

`addelem x zs` returns a list that contains all elements of `zs` as well as `x`. If `x` does not occur in `zs`, `x::zs` is returned, `zs` otherwise.

`union xs zs` returns a list containing all elements that occur in either `xs` or `zs`. Duplicates in `xs` are kept.

`diff as bs` returns a list containing all elements of `as` that do not occur in the list `bs` (in the same order as they occurred in `as`).

`minimize f zs` returns the leftmost element `z` of `zs` making the value for `(f z)` minimal. The implementation is not tail-recursive.

`uniq as` returns a list containing all elements of `as` exactly once.

Variable Handling Functions

These function might as well have been put into module `Variable` or `Schedule`. They operate on list of variables and/or schedules.

`find_block_vars sched` collects all variables that occur on the right-hand side of an assignment in an arbitrary subtree of a given schedule `sched`. The list returned is likely to contain duplicates (`uniq` is used to remove them).

`has_similar x zs` returns `true` if there is an element `z` in `zs` that satisfies the condition `Variable.similar x z`.

`overlap as bs` counts the number of overlapping variables in these two lists, i. e., the number of elements `a` in `as` that satisfy the condition `has_similar a bs`.

Reordering and Propagation

`reorder ls` rearranges a list of independent code blocks (`Par`-blocks). In the list returned, a code block with a minimal number of different variables has been put to the front and two adjacent code blocks have a maximal overlapping of variables (number of variables they have in common).

`rewrite_declarations flag a` propagates declarations of temporary variables from deeper to shallower nesting levels. The function recursively traverses an annotated schedule. If the number of temporary variables that can be declared in this block (*vl3*, i. e., the variables that are used in a sub-block but cannot be declared there, because they are used elsewhere too) exceeds a given number (`Magic.number_of_variables`) or if the declaration is forced (`flag` is `true`), the variables are declared. Otherwise, the variables are moved to *vl2* (i. e., variables that still have to be declared "outside" of this block). On the outmost nesting

level (C function) declarations of the variables that still remain to be declared are forced: As a result, `flag` is `false` on all levels but the top-level.

Generating an Annotated Schedule

`really_analyse live_at_end sched` generates a basic annotated schedule for a schedule `sched`, given which variables are still alive at the end of the block (`live_at_end`). The function operates recursively. There are 3 cases:

1. In the `Done` case, the set of variables that are alive at the end of the block equals the set of variables that are alive at the beginning of the block (no additional variables can be declared or used).

2. In the case of an assignment (`Instr(a)`), all variables that are alive at the end are also alive at the beginning. All variables occurring in the right-hand side of the assignment (the expression) are also alive at the beginning (*vl1*). There are no new declarations (*vl3* equals `[]`). The variable that the value of the expression is assigned to is already declared in this block (and it is in *vl2*).

3. Given a sequence of two blocks `a` and `b`, `Seq(a,b)`, the following property holds: Because block `a` precedes block `b`, all variables that are alive at the beginning of block `a` are also alive at the beginning of the common block `Seq(a,b)`. All variables that are alive at the beginning of block `b` are also alive at the end of block `a`.

   All variables that are used in both blocks are split into two groups: The first group consists of variables that are alive at the end of the common block (which implies that they are declared outside the common block; and they are in *vl2*). The other group is composed of variables that have a life-span that is no bigger than the life-span of the common block. They are declared locally for the common block and are in *vl3*.

`analyse live_at_end seq` recursively splits the independent parts of a schedule in multiple parts (by using `loop` und `reorder`). Single parts are transformed into an annotated schedule by `really_analyse`.

`annotate` maps a schedule to an annotated schedule. The schedule is first transformed into a basic annotated schedule by `analyse` (at the end, no variables are alive). Then the declarations are transformed again using `rewrite_declarations` into a form that is beneficial to the C compiler (size of nested blocks is specified here).

3.6 To_c: Unparsing to C

The unparser transforms the internal represenation (a dag) of the DFT into actual C code.

Description

The final phase generates C source code for the DFT calculation from an annotated schedule. The C file contains all computation-steps of the algorithm (C function), the declaration of all needed variables and constants, as well as some statistical information about "operation counts" (inserted as a comment directly into the source code file).

Data Types

```
type c_decl = Decl of string * string
type c_ast =
  | Asch of annotated_schedule
  | For of c_ast * c_ast * c_ast * c_ast
  | If of c_ast * c_ast
  | Block of (c_decl list) * (c_ast list)
  | Binop of string * expr * expr
  | Expr_assign of expr * expr
  | Stmt_assign of expr * expr
  | Comma of c_ast * c_ast
type c_fcn = Fcn of string * string * (c_decl list) * c_ast
```

Table 3.10: The data types `c_decl`, `c_ast` and `c_fcn`.

An instance of the type `c_decl` is a pair representing the declaration of a variable in C: Its first component is the type of the variable (as a string), its second component is its name (also a string).

`c_ast` is a data type for an abstract syntax tree. An AST is used for working on an abstract form of syntax without worrying about the concrete syntax of a specific programming language. The core-part of this type is — as in `Asched` — the type `annotated_schedule`. The rest of this type supports the use of control-structures (conditional execution and loops).

The type `c_fcn` describes a data type that represents a (to be generated) C function. It consists of the following parts (order is important): First the type of the return value of the function (as a string), then the name of the C function, then a list of variable declarations (for the parameters of the function) and at last the function-body (type `c_ast`).

Main Functions

There is only one function that is exported from this module, `make_c_unparser`.

Function name	*Function type*
`foldr_string_concat`	`string list -> string`
`cmdline`	`unit -> string`
`paranoid_alignment_check`	`unit -> string`
`fcn_to_expr_list`	`c_fcn -> Expr.expr list`
`count_stack_vars`	`c_fcn -> int`
`count_memory_acc`	`c_fcn -> int`
`build_fma`	`Expr.expr list -> (Expr.expr * Expr.expr * Expr.expr) option`
`count_flops_expr_func`	`int * int * int -> Expr.expr -> int * int * int`
`count_flops`	`c_fcn -> int * int * int`
`arith_complexity`	`c_fcn -> int * int * int * int * int`
`print_cost`	`c_fcn -> string`
`add_float_key_value`	`Number.number list -> Number.number -> Number.number list`
`expr_to_constants`	`Expr.expr -> Number.number list`
`extract_constants`	`c_fcn -> string`
`make_c_unparser`	`(Variable.variable -> string) -> c_fcn -> string`

Table 3.11: Function types of all functions defined in `To_c`.

Auxiliary Functions

`foldr_string_concat lss` maps a list of strings `lss` to a single string that is created by concatenating all strings together from left to right.

`cmdline` returns all arguments of the command-line as a single string, with the first argument of the command-line being the leftmost within the string.

Some compilers generate C code that stores doubles (64-bit floating point values) on an unaligned address. Although there are architectures that support this (no unalignment trap or something similar is generated), this behaviour nevertheless causes poor performance. `paranoid_alignment_check` adds the call to a macro testing the correct behaviour at run-time to the code if a certain parameter (`Magic.alignment_check`) is set.

Extracting Operation Counts

`fcn_to_expr_list fn` maps a C function (`type c_fcn`) to the list of all expressions that occur in `fn`. The order of the list returned corresponds to the execution order.

`count_stack_vars` determines the maximal number of temporary-variables that are declared at any time in the given function. The value returned is an upper bound for the number of stack variables that the code generated will need.

`count_memory_acc fn` counts all memory accesses (i.e., access to non-local data) within one given C function. Memory accesses that lie beyond the scope of programming at a source-code level (exactly predicting the result of the register allocation phase of the C compiler) are not (and can hardly be) covered here.

`build_fma` goes through a list of expressions, tries to match the various expressions against some FMA-patterns and returns if one could be found or not. The patterns used here are fairly simple, so an advanced C compiler should also be able to find all of them.

`count_flops_expr_func` counts how many mul-/add- and FMA-instructions are needed to evaluate a given expressions. It uses `build_fma` to recognize subexpressions that can be FMA-optimized.

`count_flops` iterates over a given list of expressions and applies the function `count_flops_expr_func` to count the various instructions per expression. The results are summed up and returned.

`arith_complexity` collects all operation counts returned by auxiliary functions and returns them in one compound tuple.

`print_cost fn` maps a C function to a string that describes the complexity-estimation done in `arith_complexity`. This string is inserted into the C program-text as a comment.

Extracting Constants

`add_float_key_value` adds a floating-point `key` of type `Number.number` to a list if there is not already an element `key'` in the list that satisfies `Number.equal key key'`. Otherwise the original list is returned. It is used to keep a list of constant values that are equal with regard to some measure (`Number.equal`). See Section 4.5 for more information about functions handling the abstract number type.

`expr_to_constants` accumulates all numerical constants within a given expression. It is guaranteed that there are no two elements in the list returned that are "almost equal" (defined in module `Exprdag`).

`extract_constants` maps a C function to a string containing all declarations and definitions of constants that are used within that function. The parameter

`Magic.inline_konstants` determines which definition style to use: Both the C preprocessor variant (`#define foo FFTW_KONST(...)`) and the declaration of a static variable (`static const foo=FFTW_KONST(...);`) are supported.

Unparsing the Abstract Syntax Tree

At last, `make_c_unparser var_up fn` generates the C output for the given function `fn` by utilizing a variable-unparser `var_up` (a function that maps a variable to a corresponding string).

The program-text consists of (*i*) command-line information, (*ii*) followed by some information about the arithmetic complexity estimation, (*iii*) the declaration and definition of the constants used within the C function generated and (*iv*) the source-text of the generated FFT C function.

Remarks

Because the generated C code still needs to be compiled, one has to keep in mind that the values for "operation counts" (especially the ones concerning memory accesses) are only estimates.

Chapter 4

Auxiliary Modules

In this chapter we examine the other modules that provide support for extended data types (`Complex`, `Dag`, `Expr`, `Magic`, `Number`, `Symmetry`, `Twiddle`, `Util`, `Variable`). This time the modules are listed in alphabetical order. [1]

The following sections are structured like the sections of the previous chapter into subsections named "Description", "Data Types" and "Main functions".

4.1 Complex: Operations on Complex Numbers

Description

This module is an abstraction layer for operations on complex numbers. It contains common arithmetic operations (addition, multiplication and so on) as well as operations for loading and storing complex variables.

Data Types

```
type expr =
  | CE of Exprdag.node * Exprdag.node
and variable =
  | CV of Variable.variable * Variable.variable
```

Table 4.1: The data types `expr` and `variable`.

A "complex expression" consists of two parts, a real and an imaginary part of type `Exprdag.node`. A "complex variable" consists of two "conventional" variables.

Main Functions

Complex Expressions

Almost all of the functions in this group operate on complex expressions. They implement the common operations on complex expressions.

`one` returns 1 as a complex expression.

[1] The module `Ast` has already become obsolete: It is used nowhere in `genfft` and all parts of its code are also present in `To_c`. It served as a prototype of the code that is currently used.

Function name	*Function type*
one	expr
zero	expr
inverse_int	int -> expr
times	expr -> expr -> expr
uminus	expr -> expr
swap_re_im	expr -> expr
exp	int -> int -> expr
plus	expr list -> expr
real	expr -> expr
imag	expr -> expr
conj	expr -> expr
abs_sqr	expr -> Exprdag.node
wsquare	expr -> expr
wthree	expr -> expr -> expr -> expr
sigma	int -> int -> (int -> expr) -> expr
load_var	variable -> expr
store_var	variable -> expr -> Exprdag.node list
store_real	variable -> expr -> Exprdag.node list
store_imag	variable -> expr -> Exprdag.node list
access	('a -> Variable.variable * Variable.variable) -> 'a -> variable
access_input	int -> variable
access_output	int -> variable
access_twiddle	int -> variable

Table 4.2: Function types of all functions defined in `Complex`.

`zero` returns 0 as a complex expression.

`inverse_int n` returns $1/n$ as a complex expression.

`times a b` returns $a * b$ as a complex expression.

`uminus x` returns $-x$ as a complex expression.

`swap_re_im x` swaps the real and imaginary part of the given complex expression `x` and returns the result.

`exp n i` returns the complex exponential (of root of unity).

`plus ls` adds all real and all imaginary parts of all complex expressions occurring in the given list `ls` and returns them in a complex expression.

`real x` returns the real part of a complex expression.

`imag x` returns the imaginary part of a complex expression.

`conj x` returns the conjugated complex of `x`.

`abs_sqr` returns the sum of the real part squared and of the imaginary part squared.

`wsquare w` returns the complex expression for w^2. This function assumes that $|w| = 1$ holds. Its result is equivalent to `times x x` but can be computed faster.

`wthree wn1 wn2 w` computes w^n given w^{n-1} (`wn1`), w^{n-2} (`wn2`), and w (`w`), using the identity $w^n + w^{n-2} = w^{n-1}(w + w^{-1}) = 2w^{n-1}\mathrm{Re}(w)$.

`sigma a b f` returns a list consisting of all `(f x)` with $integer(x)$ and $a \leq x < b$, i. e.,[(f a); (f (a+1)); ...; (f (b-1))].

Handling Complex Variables

`load_var cv` returns a complex expression consisting of load operations for real and imaginary parts of the given complex variable `cv`.

`store_var v e` returns a list of nodes corresponding to the store operations necessary to copy the value of the expression `e` into the complex variable `v`. Similarly, the function `store_real` only stores the real part of a complex expression, whereas `store_imag` only the imaginary part.

`access what k` returns a complex variable for accessing the element `k` of the given array `what`.

4.2 Dag: Manipulating DAGs

Description

This module includes various functions for generating, handling and transforming directed, acyclic graphs (dags, for short). Dags are often transformed into assignment lists and vice versa. Every node of the graph stands for one assignment. In addition to that, edges between the nodes indicate the dependencies between the assignments. Also, a dag contains some information (e. g., color) that is used by the scheduler to partition the graph.

Data Types

Table 4.2 shows the definition of a dag node. To simplify the construction of a graph, many attributes are declared `mutable`.

An instance of `dagnode` is a record consisting of several fields: `assigned` and `expression` are the left-hand and the right-hand side of the assignment the node corresponds to. `input_variables` is a list of all variables that occur in `expression`. The list `successors` contains all nodes that depend on this node,

```
type color = RED | BLUE | BLACK | YELLOW
type dagnode =
  { assigned             : Variable.variable;
    mutable expression   : Expr.expr;
    input_variables      : Variable.variable list;
    mutable successors   : dagnode list;
    mutable predecessors : dagnode list;
    mutable label        : int;
    mutable color        : color;
    nodeindex            : int }
and dag = Dag of (dagnode list)
```

Table 4.3: The data types `color`, `dagnode`, and `dag`.

whereas the list `predecessors` includes all nodes that this node depends on. The field `label` is used during the breadth-first search that finds the connected components of a graph. `nodeindex` is an integer value uniquely identifying every single node of the graph. The field `color` is used during graph partitioning in the scheduler.

Each node of the graph can have one of 3 colors: `RED` denotes to an input node, `YELLOW` denotes a special input node (one that loads a twiddle factor) and `BLUE` denotes an output node. `BLACK` means "not colored yet".

Main Functions

All functions implemented in this module operate on dags:

Function name	*Function type*
`node_uses`	`Variable.variable -> dagnode -> bool`
`node_clobbers`	`dagnode -> Variable.variable -> bool`
`depends_on`	`dagnode -> dagnode -> bool`
`makedag`	`(Variable.variable * Expr.expr) list -> dag`
`map`	`(dagnode -> dagnode) -> dag -> dag`
`for_all`	`dag -> (dagnode -> 'a) -> unit`
`to_list`	`dag -> dagnode list`
`find_node`	`(dagnode -> bool) -> dag -> dagnode option`
`bfs`	`dag -> dagnode -> int -> unit`

Table 4.4: Function types of all functions defined in `Dag`.

`node_uses v n` returns `true` if the node `n` depends on variable `v`.

`node_clobbers n v` returns`true` if `v` clobbers any of the input variables of `n`, i. e.,

changing the value of `v` could overwrite one of the input variables of `n`, either directly or because the input variables are aliased.

A node `b` `depends_on` another `a` if `b` uses `a` or if `b` clobbers `a`. If `b` depends on `a`, the execution of `a` must precede the execution of `b`.

`makedag alist` maps the assignment list `alist` to its corresponding dag. First the function performs the trivial mapping of the assignment list to an equivalent dag. The information of the assignment is stored and all remaining fields are initialized to default values. Then the function updates the lists of successors and predecessors according to dependency information. The resulting list of dag nodes is returned.

`map fn dag` applies the function `fn` to all dag nodes in dag, collecting the return values in the form of a dag.

`for_all dag fn` applies the function `fn` to all elements of `dag` and discards the results of the function application and returning `()`. It only purpose is to cause some side effects, such as resetting the color of all nodes in a dag.

`to_list dag` maps a dag to a list of dag nodes.

`find_node f dag` tries to find a node `n` in the given graph that satisfies the condition `(f n)`. In case of failure, it returns `None`. `Some n` otherwise.

`bfs dag node init_label` performs breadth-first search to mark all nodes that are reachable from a given start node in the graph. It is used to find connected components of a dag.

4.3 Expr: Representation of Expressions

Description

Expressions are used for the internal representation of algebraic terms.

An assignment consists of a variable `v` and an expression `e`. It denotes the assignment of an expression `e` to a variable `v`.

Data Types

Table 4.5 defines expressions. Each expression encodes one operator, the nodes children are the operator's expressions.

Integer expressions are only used for loop counters in the generated C code. Unary minus help storing only positive numbers which helps increase performance.

```
type expr =
  | Num of Number.number
  | Var of Variable.variable
  | Plus of expr list
  | Times of expr * expr
  | Uminus of expr
  | Integer of int
and assignment = Assign of Variable.variable * expr
```

Table 4.5: The data type `expr`. An expression is either a floating-point constant value `k`, a variable `v`, a list of expressions `ps` that are added, a product of two expressions `e1` and `e2`, the negation of an expression `m`, or an integer `i`.

Function name	*Function type*
`find_vars`	`expr -> Variable.variable list`

Table 4.6: Function types of all functions defined in `Expr`.

Main Functions

There is only one function exported and implemented in this module:

`find_vars e` collects the list of variables of all kind (input, output, twiddle, named or temporary) that occur within the given expression `e`. Variables can occur more than once in this list. The list is in no particular order.

4.4 Magic: Global Parameters

We now give an explicit list of the "magic"-parameters that are used in the modules described earlier. Table 4.7 presents magic parameters of type `int`.

Module	*Name*	*Description*
`Asched`	`number_of_variables`	minimal number of variables per declaration
`Fft`	`rader_list`	list of transform lengths such that use of Rader's algorithm is forced
`Fft`	`rader_min`	minimal transform length such that use of Rader's algorithm is considered

Table 4.7: "Magic" parameters of type `int` or `int list`.

Table 4.8 shows magic parameters of type `bool`.

Module	*Name*	*Description*
`Fft`	`athenafft`	use experimental generator routines
`To_c`	`alignment_check`	check alignment of `double` on the stack
`Fft`	`alternate_convolution`	internal flag of Rader generator
`Exprdag`	`collect_common_inputs`	internal flag of the CSE
`Exprdag`	`collect_common_twiddle`	internal flag of the CSE
`Exprdag`	`enable_fma`	FMA extraction flag
`Exprdag`	`enable_fma_expansion`	FMA extraction flag
`To_c`	`inline_konstants`	declare constants as `static const`
`Exprdag`	`inline_loads`	do not extract input variables in CSE
`Exprdag`	`inline_single`	inline dag node that it is used only once
`Schedule`	`loopo`	parameter for graph-partitioning
`Twiddle`	`twiddle_policy`	specifies the twiddle policy to be used
`Util`	`verbose`	display status information on `stderr`
`Twiddle`	`wsquare`	use `Complex.wsquare` instead of `Complex.times` when calculating w^{2n}

Table 4.8: "Magic" parameters of type `bool`. The flags `enable_fma` and `enable_fma_expansion` control the utilization of "fused multiply-add" instructions. These instructions are supported on some microprocessor architectures and allow an expression of type $\pm xy \pm z$ to be evaluated with only one instruction.

4.5 Number: An Abstract Number Type

Description

The generator keeps track of numeric constants in symbolic expressions using the abstract number type defined in this module.

The implementation of the `number` type uses arbitrary-precision arithmetic from the built-in `Num` package in order to maintain an accurate representation of constants. Consequently, the generator can take advantage of arbitrary-precision floating point values, thereby taking advantage of the full precision available on current and future machines.

Since the `Num` package only supplies rational arithmetic, a routine to compute the (irrational) complex roots of unity is provided. The arbitrary-precision operations in `Num` look like the normal operations except that they have an appended slash (e. g.,`+/`, `-/`, ...).

Data Types

```
type number =
  | N of Num.num * float
```

Table 4.9: The data type `number`. An instance of this type consists of an arbitrary-precision rational (first component) and a floating-point value (for efficiency reasons). To limit space-consumption, all operators immediately round the results to `precision` digits.

Defined Constants

Name	*Type*	*Description*
`precision`	`int`	decimal digits of precision to work with
`print_precision`	`int`	precision to use when printing numbers
`inveps`	`Num.num`	$10^{\text{precision}}$
`epsilon`	`Num.num`	$10^{-\text{precision}}$
`epsilonsq`	`Num.num`	epsilon^2
`epsilonsq2`	`Num.num`	$100 * \text{epsilon}^2$
`pinveps`	`Num.num`	$10^{\text{print\_precision}}$
`pepsilon`	`Num.num`	$10^{-\text{print\_precision}}$
`zero, one, two, mone`	`number`	0, 1, 2, -1
`twopi`	`float`	2π

Table 4.10: Constants defined in the module `Number`.

Main Functions

Function name	*Function type*
equal	number -> number -> bool
is_zero	number -> bool
is_one	number -> bool
is_mone	number -> bool
is_two	number -> bool
negative	number -> bool
greater	number -> number -> bool
almost_equal_cnum	Num.num pair -> Num.num pair -> bool
makeNum	Num.num -> number
round	Num.num -> Num.num
of_int	int -> number
unparse	number -> string
to_string	number -> string
to_float	number -> float
mul	number -> number -> number
div	number -> number -> number
add	number -> number -> number
sub	number -> number -> number
negate	number -> number
csub	Num.num pair -> Num.num pair -> Num.num pair
cdiv	Num.num pair -> Num.num -> Num.num pair
cmul	Num.num pair -> Num.num pair -> Num.num pair
csqr	Num.num pair -> Num.num pair
cabssq	Num.num pair -> Num.num
cconj	Num.num pair -> Num.num pair
cinv	Num.num pair -> Num.num pair
ipow_cnum	Num.num pair -> int -> Num.num pair
primitive_root_of_unity	int -> Num.num pair
cexp	int -> int -> number * number

Table 4.11: Function types of all functions defined in `Number`.

There are three groups of functions implemented in this module: functions for comparing numbers, functions for conversion of some data from/to the number type and functions performing some arithmetic operation on numbers. Each of these groups is described in the following:

Comparing Numbers

`equal` compares two arbitrary precision big numbers for equality. They are considered equal if (*i*) their absolute difference (error) or (*ii*) their relative error is less than $10^{-\texttt{print\_precision}}$.

Functions named `is_...` compare a given number with a constant. They return `true` if the two numbers are equal w.r.t. the definition given above.

`negative x` returns `true` if the `x` is less than zero.

`greater a b` returns `true` if `(negative (b-a))` holds.

`almost_equal_cnum c1 c2` compares two complex numbers for equality. They are considered almost equal if the sum of the squares of the component-wise difference is less than `epsilonsq2`.

Converting Numbers from/to Other Data Types

`makeNum x` packs a given value `x` and its floating point representation into a `number` structure and returns it.

`round x` rounds the number to have at most `precision` digits precision.

`of_int x` maps an integer to a number.

`unparse x` finds an appropriate decimal representation of a given number `x` and returns it as a string.

`to_string n` converts a number to a string using the default output precision `print_precision`.

`to_float n` converts a number to a floating-point number, i.e., it simply strips of the arbitrary-precision integer from the pair `n`.

Operations on Complex Numbers

Note that, in the following computations, it is important to round to precision epsilon after each operation. Otherwise, since the `Num` package uses exact rational arithmetic, the number of digits quickly blows up.

The functions `mul`, `div`, `sub`, and `add` take two numbers as arguments, perform a (multiplication, division, subtraction or addition) operation, round the result immediately and return it.

`negate x` returns $-x$.

`csub`, `cdiv`, `cmul`, `csqr`, `cabssq`, `cconj`, and `cinv` perform the following operations on one or more complex numbers: subtraction of two complex numbers, division of a complex number by a scalar, multiplication of two complex numbers,

component-wise squaring of a complex number, summing the absolute square, building the conjugate complex and inverting a complex number.

`ipow_cnum x n` generates x^n where `x` is a complex number and `n` is an integer.

`primitive_root_of_unity n` find the n-th (complex) primitive root of unity by Newton's method.

4.6 Symmetry: Data Symmetries

Description

A *symmetry* is a rule defining some form of equality between (input, output or intermediate) array elements with different indices. Exploiting symmetries enables `genfft` to create transforms that perform fewer memory operations or that use a more compact form of data-representation, aiming at better overall performance.

One example of data symmetries already occurred in the Chapter 2. In that chapter we gave an example of how `genfft` handles (one-dimensional) real transforms and how a *hermitian* array is represented (as a half-complex array).

Data Types

```
type symmetry =
  { apply : int -> (int -> Complex.expr) -> int -> Complex.expr;
    store : int -> (int -> Complex.expr) -> int -> Exprdag.node list;
    osym  : symmetry;
    isym1 : symmetry;
    isym2 : symmetry }
```

Table 4.12: The data type `symmetry`.

Symmetries are encoded as symmetries of the *input*. A symmetry determines (*i*) the symmetry of the output (`osym`), (*ii*) symmetries at intermediate stages of divide and conquer or Rader (`isym1` and `isym2`).

`store n f i` returns a list of `Exprdag.node`s that represent a store-instruction in symmetric data. `n` is the size of the DFT. `f` maps an array index to the corresponding element.

`apply n f i` is used for getting an element equivalent to the i-th element of f (with regard to the symmetry; the function can be seen as an array).

4.7 Twiddle: Loading Policies of Twiddle Factors

Description

There are many ways of determining twiddle factors. The various policies are implemented in this module. In this chapter the "Data Types" section is missing because there are no new data types defined in this module.

Main Functions

Function name	*Function type*
`square`	`Complex.expr -> Complex.expr`
`twiddle_policy_load_all`	`('a -> int -> 'b -> Complex.expr) * (int -> int) * (int -> int list)`
`twiddle_policy_load_odd`	`('a -> int -> (int -> Complex.expr) -> Complex.expr) * (int -> int) * (int -> int list)`
`twiddle_policy_iter`	`('a -> int -> (int -> Complex.expr) -> Complex.expr) * ('b -> int) * ('c -> int list)`
`twiddle_policy_square1`	`('a -> int -> (int -> Complex.expr) -> Complex.expr) * ('b -> int) * ('c -> int list)`
`twiddle_policy_square2`	`('a -> int -> (int -> Complex.expr) -> Complex.expr) * ('b -> int) * ('c -> int list)`
`twiddle_policy_square3`	`('a -> int -> (int -> Complex.expr) -> Complex.expr) * ('b -> int) * ('c -> int list)`
`twiddle_policy`	`unit -> ('a -> int -> (int -> Complex.expr) -> Complex.expr) * (int -> int) * (int -> int list)`

Table 4.13: Function types of all functions defined in `Twiddle`.

`square x` returns the square of a given complex expression `x`. Depending on a global parameter (`magic.use_wsquare`) either the function `Complex.times` or `Complex.wsquare` is used for calculating the square.

`twiddle_policy` returns a function representing the twiddle loading policy, i.e., loading and/or computing twiddle factors, depending on a global parameter (`Magic.twiddle_policy`).

Every policy-function returns a 3-tuple representing the loading policy itself, the number of twiddle factors loaded as well as a list indicating the order of the loads. This information is inserted in the description of the codelet in the C code source file.

Most functions in this module (except for `twiddle_policy_load_all`) are experimental that are usually not used. All implemented policies are listed in Table 4.14:

Name	*Rule*	*#*	*Order*
load_all	Load all twiddle factors.	$n-1$	`[1;2;...;n]`
load_odd	If n is even, $w^n = (w^{n/2})^2$. If n is odd, load it.	$n/2$	`[1;3;...;n]`
iter	$w^n = w^{n-1}w$.	1	`[1]`
policy1	If n is even, $w^n = (w^{n/2})^2$. If n is odd, $w^n = w^{n-1}w$.	1	`[1]`
policy2	If n is even, $w^n = (w^{n/2})^2$. Else, compute w^n from w^{n-1}, w^{n-2}, and w.	1	`[1]`
policy3	If n is even, $w^n = (w^{n/2})^2$. If n is odd, $w^n = w^{floor(n/2)}w^{ceil(n/2)}$.	1	`[1]`

Table 4.14: Twiddle factor loading/computing policies. By default all twiddle factors are loaded.

4.8 Util: Generic Functions on Integers and Lists

Description

In a functional programming language, very general functions can be written that operate on generic data type (especially functions that use homogenous lists). This modules collects many functions of this kind. It does not define new data types.

Main Functions

`identity` is the identity function. The same function was called `return` in `Schedule`.

`(@@)` is a function composing two other functions. `(@@) f g x` first applies the function `g` to `x` and then applies `f` to the result, returning the result of the latter function application.

Function name	*Function type*
`invmod`	`int -> int -> int`
`gcd`	`int -> int -> int`
`lowest_terms`	`int -> int -> int * int`
`find_generator`	`int -> int`
`pow_mod`	`int -> int -> int -> int`
`suchthat`	`int -> (int -> bool) -> int`
`forall`	`('a -> 'b list -> 'b list) -> int -> int -> (int -> 'a) -> 'b list`
`sum_list`	`int list -> int`
`min_list`	`int list -> int`
`max_list`	`int list -> int`
`count`	`('a -> bool) -> 'a list -> int`
`filter`	`('a -> bool) -> 'a list -> 'a list`
`remove`	`'a -> 'a list -> 'a list`
`cons`	`'a -> 'a list -> 'a list`
`null`	`'a list -> bool`
`(@@)`	`('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`
`forall_flat`	`int -> int -> (int -> 'a list) -> 'a list`
`identity`	`'a -> 'a`
`for_list`	`'a list -> ('a -> 'b) -> unit`
`minimize`	`('a -> 'b) -> 'a list -> 'a option`
`find_elem`	`('a -> bool) -> 'a list -> 'a option`
`info`	`string -> unit`

Table 4.15: Function types of all functions defined in `Util`.

Integer-Based Functions

The following functions are mainly used for handling multiplicative groups:

`gcd n m` determines and returns the greatest common divisor of the integers `n` and `m` using Euclid's algorithm.

`find_generator p` tries to find a generator element of the multiplicative group modulo `p` and returns it.[2] A generator exists if `p` is prime.

`pow_mod x n p` raises `x` to a power of `n` modulo `p`. If `n` is negative an expection (`Negative_Power`) is raised.

`invmod n m` finds and returns the inverse element of `n` modulo `m`. It requires `m` and `n` to be relatively prime.

[2] A generator element of a finite multiplicative group modulo p is an element x, such that $x^0 = x^{p-1} \; mod \; p = 1$ holds, with all $x^i \; mod \; p$ (with $1 \leq i \leq p$) being distinct and not equal to zero.

`lowest_terms n m` normalizes the rational number m/n (which has to be smaller than 1.0) and returns the result.

`suchthat a pred` finds and returns an integer value `x` that is greater than or equal to `a`, such that `x` satisfies the condition `pred x`. Great care has to be taken when using this function (risk of non-termination!).

List Based Functions

`sum_list ls` calculates the sum of the given list of integers and returns it.

`max_list ls` determines the biggest integer in the list `ls` and returns it.

`min_list ls` returns the smallest integer element in the given list `ls`.

`count pred ls` counts the number of elements in the given list `ls` that satisfy the condition `pred`.

`filter pred ls` returns a list only containing elements satisfying the given condition `pred`.

`remove elem ls` returns a list containing all elements of `ls` that are not equal to `elem`.

`cons a b` is syntactic sugar for `(a::b)`. `::` is the infix operator predefined in ML for constructing a list. `a` and `b` are the head and the tail of the list (also called "car" and "cdr").

`null ls` returns `true` if the given list is empty.

`for_list ls f` iterates over the given list `ls` from left to right and applies the function `f` to each element throwing away the return value of the function. It can be used to do some side effects on each element of a list.

`minimize f ls` looks for the leftmost element `x` minimizing the value of `f x` and returns it (`Some x`). If no element is contained in the given list, `None` is returned.

`find_elem pred xs` returns the leftmost element `x` of the given list that satisfies the condition `pred x`. In case of success `Some x` is returned, `None` otherwise.

`forall c a b f` returns a data structure containing all `(f x)` with $integer(x)$ and $a \leq x \leq b$ combined by the function `c`.

`forall_flat a b f` returns `(f a) @ (f (a+1)) @ ... @ (f b)`.

Remaining Functions

The function `info` displaying status information (it is used to indicate progress and can be used to ease debugging). It is primarily used in modules `Genfft` and `Exprdag`. `info str` displays the string `str` on the channel `stderr` if `Magic.verbose` equals `true`. In addition to the string specified, some status information (runtime of the process and process-id) is printed out, too.

4.9 Variable: Variable Handling and Unparsing

Description

Variables play an important role throughout the entire runtime of `genfft`. The various variable types supported are defined in Table 4.9.

Data Types

```
type array =
  | Input
  | Output
  | Twiddle
and variable =
  | Temporary of int
  | Named of string
  | RealArrayElem of (array * int)
  | ImagArrayElem of (array * int)
```

Table 4.16: The data types `array` and `variable`.

An instance of the data type `array` refers to either the input-data (`Input`), the output-data (`Output`) or the twiddle-factors (`Twiddle`).

A variable is either a temporary variable with some index `t` (`Temporary(t)`), a variable with a fix name `s` (`Named(s)`), the real part of the `n`-th element of the array `arr` (`RealArrayElem(arr, n)`) or the imaginary part of the `m`-th element of the array `arr` (`ImagArrayElem(arr, m)`).

Main Functions

Basic Functions

These functions are all about generating variables and about testing various properties of one (or many) variables.

`make_temporary ()` generates a new temporary variable with a unique index and returns it.

`make_named n` generates a named variable with name `n`.

`is_temporary v` returns `true` if `v` refers to a temporary variable.

`is_input v` returns `true` if `v` refers to an input variable.

`is_output v` returns `true` if `v` refers to an output variable, that is if it is either a real or an imaginary array element (`RealArrayElem(Output,_)` or `ImagArrayElem(Output,_)`).

Function name	*Function type*
make_temporary	unit -> variable
make_named	string -> variable
is_temporary	variable -> bool
is_output	variable -> bool
is_input	variable -> bool
is_twiddle	variable -> bool
clobbers	variable -> variable -> bool
real_imag	variable -> variable -> bool
increasing_indices	variable -> variable -> bool
access_input	int -> variable * variable
access_output	int -> variable * variable
access_twiddle	int -> variable * variable
same	'a -> 'a -> bool
similar	variable -> variable -> bool
hash	variable -> int
unparse_index	string -> string option -> int -> string
default_unparser	variable -> string
make_unparser	string * string option -> string * string option -> string * string option -> variable -> string

Table 4.17: Function types of functions defined in `Variable`.

`is_twiddle v` returns `true` if `v` refers to a twiddle factor.

`same a b` returns `true` if `a` and `b` are structurally equal (`=`).

`hash v` returns a hash value (that is mainly based on the variable type) for the given variable `v`.

`similar v1 v2` determins whether two variables, `v1` and `v2` are similar. Two variables are similar if they are equal. They are also similar if `v1` refers to the real and `v2` to the imaginary part (or vice versa) of the same array and if they have the same index. It is used for raising cache performance.

`clobbers v1 v2` returns `true` if `v1` is an output variable, `v2` is an input variable, and if—assuming that input and output arrays are exactly aliased (i. e., in place modification of data)— storing a value into `v1` would destroy the existing value of `v2`.

`real_imag a b` returns `true` if `a` is the real part and `b` the imaginary of the same array element.

`increasing_indices a b` returns `true` if `a` and `b` are elements of the same array, and `a` has smaller index. This function was part of an experiment for testing caching-behavior and it not currently used.

`access arr k` returns a pair consisting of a real and an imaginary part referring to the same array `arr` and having the same index `k`. `access_input k` returns a pair referring to the input, `access_output k` returns a pair referring to the output and `access_twiddle k` returns a pair referring to a twiddle factor.

Unparsing Support

`make_unparser in out tw v` unparses a given variable `v` using the information (regarding names and strides) contained in `in` (used for input variables), `out` (used for output variable) and `tw` (used for twiddle factors).

`default_unparser v` unparses named and temporary variables to a string that it returns. Otherwise it fails raising the exception `failure("attempt to unparse unknown variable")`.

`unparse_index n st k` maps an array name `n`, stride information `st` and the index of the element to access to a string of the form `n[index]`. If there is a stride, `index` is a stratified form of `st*k`. Otherwise `index` is `k`.

Chapter 5

Supplemental Information

In this chapter we provide some additional information that is not crucial for understanding the basic mechanisms of `genfft`.

5.1 Exceptions

In this section we look at the way `genfft` handles abnormal program conditions.

The language CAML defines a clear way of handling abnormal program situations (see [Ler98]). These mechanism of raising and handling exceptions is used extensively throughout the standard library of CAML.

FFTW (`genfft`) uses this mechanism in order to react in a clean way to a function called with wrong arguments, i.e., to handle argument errors that the type system cannot handle easily. This use of exceptions is comparable to using assertions in C.

In some cases equivalent functions are implemented using the exception-handling approach as well as the conventional method of returning a special value in case of an error, e.g., one implementation of `minimize` throws a `Failure("minimize")` exception when applied to an empty list, whereas the other implementation returns a value of type `Option` indicating success or failure.

The code of `genfft` itself does not "react" to exceptions. It ignores them, causing the ML runtime system to issue an error string (as a debug message) on `stderr`.

Almost all of the exceptions used have the form `Failure(...)`, only two have other (more explicit) names. Table 5.1 lists all exceptions used in `genfft`.

5.2 GENFFT Changelog

In FFTW 2.1.3 the following modifications have been made:

- The module `Ast` was removed from the `FFTW` distribution.
- The data type `Number.number` has been simplified by removing the floating-point number component.

Module	*Exception name*
Asched	Failure("minimize")
Asched	Failure("really_analyze")
Asched	Failure("loop")
Exprdag	Failure("bug in dualExpressionM")
Exprdag	Failure("with_temp_maybeM")
Genfft	Failure("too many arguments")
Genfft	Failure("one of -notwiddle, ... must be specified")
Schedule	Failure("connected")
Symmetry	Failure("middle_hc2hc_backward_sym")
Symmetry	Failure("final_hc2hc_backward_sym")
Symmetry	Failure("symmetric_sym")
Symmetry	Failure("anti_symmetric_sym")
Symmetry	Failure("realeven2_input_sym")
Symmetry	Failure("realodd2_input_sym")
Util	No_Generator
Util	Negative_Power
Variable	Failure("attempt to unparse unknown variable")
Variable	Failure("trying to access imaginary part of real input")
Variable	Failure("trying to access imaginary part of real output")
Variable	Failure("trying to access real part of imaginary output")

Table 5.1: All exceptions used in `genfft`.

Bibliography

[ABF+99] M. Auer, R. Benedik, F. Franchetti, H. Karner, P. Kristöfel, R. Schachinger, A. Slateff, and C. W. Ueberhuber. Performance Evaluation of FFT Routines. Technical Report AURORA TR1999-05, Technical University of Vienna, Jan 1999.

[CT65] J. W. Cooley and J. W. Tukey. An algorithm for the machine computation of the complex Fourier series. *Mathematics of Computation*, 19:297–301, Apr 1965.

[Duf65] R. J. Duffin. Topology of Series-Parallel Networks. *Journal of Mathematical Analysis and Applications*, 10:303–318, 1965.

[DV90] P. Duhamel and M. Vetterli. Fast Fourier transforms: a tutorial review and a state of the art. *Signal Processing*, 19:259–299, Apr 1990.

[FJ] M. Frigo and S. G. Johnson.
The FFTW web page. `http://www.fftw.org`.

[FJ97] M. Frigo and S. G. Johnson. The fastest Fourier transform in the West. Technical Report MIT-LCS-TR728, MIT Lab for Computer Science, Sep 1997.

[FJ98] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, May 1998.

[FLPR99] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious algorithms. *In the 40th Annual Symposium on Foundations of Computer Science (FOCS '99)*, Oct 1999.

[Fri99] M. Frigo. A Fast Fourier transform compiler. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1999.

[GDT+99] M. Galassi, J. Davies, J. Theiler, B. Gough, R. Priedhorsky, G. Jungman, and M. Booth. *GNU Scientific Library—Reference Manual*, 1999.

[Ler98] X. Leroy. The Objective Caml system release 2.00. Technical report, Institut National de Recherche en Informatique et en Automatique, Aug 1998. `http://caml.inria.fr`.

[Mac92] P. A. MacMahon. The Combination of Resistances. *The Electrician*, April 1892.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT, August 1990.

[OS89] A. V. Oppenheim and R. W. Schafer. *Discrete-time Signal Processing*. Prentice-Hall, Englewood Cliffs, 1989.

[Rad68] C. M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proc. of the IEEE*, 56:1107–1108, Jun 1968.

[RS42] John Riordan and C. E. Shannon. The Number of Two-Terminal Series-Parallel Networks. *Journal of Mathematics and Physics*, 21:83–93, 1942.

[SJHB87] H. V. Sorensen, D. L. Jones, M. T. Heideman, and C. S. Burrus. Real-valued fast Fourier transform algorithms. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(6):849–863, June 1987.

[Swa82] P. N. Swarztrauber. Vectorizing the FFTs. *Parallel Computations*, pages 51–83, 1982. G. Rodrigue ed.

[Val78] J. Valdes. *Parsing Flowcharts and Series-Parallel Graphs*. PhD thesis, Stanford University, December 1978. STAN-CS-78-682.

[Wad92] P. Wadler. The Essence of Functional Programming. In *Proceedings of the 19th Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 19–22, 1992. ACM Press.

[Wad97] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, September 1997.