A Haskell Compiler for Signal Transforms

Geoffrey Mainland  
Computer Science  
Drexel University, USA  
mainland@drexel.edu

Jeremy Johnson  
Computer Science  
Drexel University, USA  
jjohnson@cs.drexel.edu

Abstract
Building a reusable, auto-tuning code generator from scratch is a challenging problem, requiring many careful design choices. We describe HSpiral, a Haskell compiler for signal transforms that builds on the foundational work of Spiral. Our design leverages many Haskell language features to ensure that our framework is reusable, flexible, and efficient. As well as describing the design of our system, we show how to extend it to support new classes of transforms, including the number-theoretic transform and a variant of the split-radix algorithm that results in reduced operation counts. We also show how to incorporate rewrite rules into our system to reproduce results from previous literature on code generation for the fast Fourier transform.

Although the Spiral project demonstrated significant advances in automatic code generation, it has not been widely used by other researchers. HSpiral is freely available under an MIT-style license, and we are actively working to turn it into a tool to further both our own research goals and to serve as a foundation for other research groups’ work in developing new implementations of signal transform algorithms.

CCS Concepts  
• Software and its engineering → Domain specific languages; Source code generation;  
• Mathematics of computing → Mathematical software;

Keywords  
domain specific languages, code generation, code optimization, rewrite rules, fast Fourier transform

ACM Reference Format:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

1 2017 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5524-7/17/10...
$15.00
https://doi.org/10.1145/3136040.3136056

1See www.spiral.net for a wide range of references.

1 Introduction
The Spiral project [35, 36] has demonstrated the value of a high-level mathematical domain specific language for designing, implementing and optimizing implementations of signal processing algorithms. Initially, Spiral used SPL (signal processing language) for representing and implementing signal transforms, such as the fast Fourier transform (FFT), fast trigonometric transforms, and wavelet transforms. Such transforms are linear, and SPL used a mathematical language to encode algorithms for computing linear transformations as sparse structured matrix factorizations. During the period prior to the development of Spiral, there was extensive research, summarized in the books by Van Loan [41] and Tolimieri et al. [40], into the use of algebraic methods for describing and deriving signal transforms. SPL and its compiler were an attempt to directly implement these mathematical descriptions. In addition, Spiral used high-level mathematical rules to automatically derive algorithms, also defined by SPL terms, for computing transforms. Using these rules, many algorithm variants could be generated, and various search methods were used to select high-performance implementations. Spiral differed from other projects that automatically tune mathematical kernel implementations, such as FFTW [15] and ATLAS [7], through its use of an extensible language for describing algorithms. The initial implementation focused on straight-line code and used special cases to obtain efficient loop code. Moreover, the code produced was restricted to fixed size transforms.

Subsequently, considerable research has been undertaken by the Spiral project to improve performance and extend the range of algorithms and hardware platforms supported by Spiral. Extensions have been made to better support loops [11], vectorization [13], parallelism [12], and hardware generation [32]. The ability to generate code for arbitrary size inputs was developed [43], and the language SPL was generalized to operator language (OL) [8] to support a wider range of algorithms. More recently, an effort has been initiated to simultaneously generate code and certificates validating its correctness [9]. The performance of the code generated by Spiral has been demonstrated through comparisons to the state-of-the-art and by its use by Intel in their Math Kernel Library (MKL) and Integrated Performance Primitives (IPP) library. Many of these improvements have been made incrementally to demonstrate the effectiveness of the techniques.
developed and have not been generalized and made easy to use. Consequently, Spiral has not been widely used by other researchers. Moreover, since Spiral has been developed in a language not designed to support DSLs, it has not fully benefited from advances in the theory and practice of domain specific languages.

The effort reported in this paper is an attempt to revisit the design and implementation of Spiral with an emphasis on usability and incorporation of state-of-the-art tools for the design and implementation of DSLs. In particular, we provide a Haskell implementation of an extension of the original SPL language with the goal of a simpler implementation that is easy for language designers to extend and experiment with along with being easy for algorithm designers to use. We describe our initial implementation and its use to explore a recently discovered improvement to the split-radix algorithm that reduces the number of real multiplications required by FFTs of size $2^n$ [21]. Our implementation further reduces the number of multiplications, and we can explain the trade-off between local code optimizations [14, 25] and a high-level mathematical description.

What is the benefit of reimplementing SPL? Because HSpiral is written in Haskell, new users can rely on a much larger active user base and set of learning materials than GAP [39], the system in which Spiral was originally implemented, can provide. Our use of language techniques that are well-known in the Haskell community makes it easier for users to extend the implementation. Haskell’s features—especially its type system—also allow us to design abstractions that reduce code duplication, simplifying many aspects of the implementation. Finally, unlike Spiral, HSpiral is freely available under an MIT-style license. Concretely, our contributions are as follows:

- We implement an extended version of SPL in Haskell as a compiler from SPL expressions to C (Section 3). We demonstrate that this implementation is extensible along several dimensions (Section 4) and that it has reasonably good performance (Section 5).
- We show how, using rewrite rules, one can derive FFT implementations that match the operation count of the split-radix discrete Fourier transform (DFT) from the decimation-in-frequency DFT. This explains results from prior work, including the operation counts reported by Kiselyov and Taha [25] as well as the “DAG-reversal” trick utilized by FFTW [14].
- We formulate the improved split-radix rule given by Johnson and Frigo [21] in SPL. Our compiler produces code with fewer operations than their implementation. We speculate that this is due to our use of cyclotomic polynomials to find additional simplifications and common subexpressions.

2 https://github.com/mainland/hspiral

2 The SPL Language

The SPL language [45] is a DSL for representing and implementing fast algorithms for computing linear transformations, $y = Ax$. It was motivated by three observations:

1. Fast signal processing algorithms, such as the fast Fourier transform (FFT), could be expressed as a product of sparse structured matrices [40, 41].
2. Matrix formulations provided a convenient way of describing and understanding the many variants of these algorithms [40, 41].
3. The mathematical description of these algorithms in terms of matrices and matrix operators, such as the tensor product, had a natural interpretation in terms of operations on high-performance computing platforms [20].

SPL programs are symbolic matrix expressions built from matrix constructors, families of parameterized matrices, and various matrix operators such as composition, direct sum, sum, transpose, and tensor product. Matrix constructors allow a list of elements, both dense and sparse, a list of SPL expressions, which can be used to construct block matrices, and a function mapping indices to element expressions or SPL expressions. Similar constructors are provided for special classes of matrices such as permutations and diagonal matrices. These constructors are more general than those in the original version of SPL [45], which used a code template mechanism to define parameterized matrices. The new constructors are similar to the constructs for diagonal and permutation matrices in E-SPL [11] and are designed to make it easier to define and construct matrices. In this spirit, we also allow indexed versions of the operators, which further facilitate the definition of families of matrices.

Every SPL expression can be interpreted as a matrix of a fixed dimension whose elements come from some specified domain. Any two SPL expressions can be checked for equivalence by computing the corresponding matrices and checking for equality. The SPL compiler translates SPL expressions into programs that can be applied to an input vector of a given type and produce an output vector of the same type. The program produced by the SPL compiler must have the property that given an SPL expression $\mathcal{A}$, it produces a program that, when applied to a vector $x$, returns the vector $y = Ax$, where $A$ is the matrix interpretation of $\mathcal{A}$.

To illustrate SPL, let $n$ be a positive integer, $I_n$ be the $n \times n$ identity matrix, and $\text{DFT}_n(\omega) = [\omega^{|d| j}]_{0 \leq i, j < n}$ be the $n$-point DFT matrix, where $\omega$ is a primitive $n$th root of unity. Assume $d|n$ and let $T^d_n = \bigoplus_{e|d} W_{n/e}(\omega)^e$, where $W_{n/e}(\alpha) = \text{Diag}(1, \alpha, \ldots, \alpha^{e-1})$ and $L^d_n$ is the permutation matrix whose $(n/d) + i, id + j$ element for $0 \leq i < d$ and $0 \leq j < n/d$ is equal to 1. Further, let $A$ be an $m_1 \times n_1$ matrix and $B$ be an $m_2 \times n_2$ matrix. Then $A \circ B$, the Kronecker or tensor product, is the $m_1 m_2 \times n_1 n_2$ block matrix $[a_{ij}B]$. 
The following SPL expression is a factorization of $\text{DFT}_4 = \text{DFT}_{4}(i)$. In this example, we assume that the element domain is the complex numbers and that the imaginary unit $i$ is chosen as a primitive 4-th root of unity.

$$
\text{DFT}_2 \otimes \text{I}_2 \otimes \text{DFT}_2 \otimes \text{I}_2 \\
= \begin{bmatrix}
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0
\end{bmatrix}
= \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

The factorization of $\text{DFT}_4$ provides a more efficient means of computing $\text{DFT}_4$ if we apply each of the matrices one after the next to the input vector. In this example, the number of arithmetic operations is reduced from 12 to 8.

The previous example can be generalized into a parameterized rule, which is the basis of the FFT algorithm. The rule comes in two flavors, where the second is the transpose of the first, which holds since the FFT is symmetric.

**Theorem 2.1 (Cooley-Tukey Rule).**

\[
\begin{align*}
\text{DFT}_{rs}(\omega) &= (\text{DFT}_r(\omega^s) \otimes \text{I}_s) T_s^{rs}(\omega) (\text{I}_r \otimes \text{DFT}_s(\omega^s)) L_r^{rs} \\
\text{DFT}_{rs}(\omega) &= L_s^{rs}(\text{I}_r \otimes \text{DFT}_s(\omega^s)) T_r^{rs}(\omega) (\text{DFT}_r(\omega^s) \otimes \text{I}_r)
\end{align*}
\]

(Decimation in Frequency)

where $\omega$ is a primitive $n = rs$-th root of unity.

The proof of this rule [20, 40, 41] only depends on properties of roots of unity and holds over any field that has the necessary roots of unity. For example, the previous example holds for $\mathbb{Z}/p$, the integers mod $p$, with $p = 17$ where $\omega = 13$ is a primitive 4-th root of unity.

$$
\text{DFT}_4(13) = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 13 & 16 & 4 \\
1 & 16 & 1 & 16 \\
1 & 4 & 16 & 13
\end{bmatrix}
= \begin{bmatrix}
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 16 & 0 \\
0 & 1 & 0 & 16
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0
\end{bmatrix}
$$

Algorithms for computing the DFT can be obtained by repeatedly applying the Cooley-Tukey rule until in the base case where the DFT is replaced by the matrix from its definition. If the rule is repeatedly applied with $r = 2$, the resulting algorithm is the standard radix 2 FFT. Iterative versions of these algorithms can be concisely specified using indexed composition.

$$
\text{DFT}_{2^k} = \left\{ \prod_{i=0}^{k} (\text{I}_{2^i} \otimes \text{DFT}_2 \otimes \text{I}_{2^{k-i}}) (\text{I}_{2^i} \otimes T_{2^k}^{2^{k-i}}) \right\} R_{2^k}
$$

where $R_{2^k}$ is the bit-reversal permutation [20]. Additional FFT rules are available for other FFT algorithms such as split-radix [46], Good-Thomas [16, 17], Rader [37], and Bluestein’s algorithm [2], and all can easily be expressed in SPL.

### 3 Implementation

The HSprial implementation is built from several independent components that are not tied to the SPL language and could each be reused in other situations where code generation is an appropriate tactic. At a high-level, these components can be grouped by functionality as follows:

- An embedded DSL for representing pure mathematical expressions.
- The SPL embedded DSL.
- A library for array computation that supports symbolic array indices.
- An embedded DSL for representing impure computations.
- A C back-end that translates the computation DSL to C.
- A library for search.

Our three DSLs are deeply embedded—they represent the structure of DSL terms using an explicit abstract syntax data type. However, the users of these DSLs can write standard Haskell without worrying about this detail, since abstract syntax terms are constructed automatically by our libraries. While deep embeddings and other techniques we used in developing HSprial are standard, the unique requirements of the SPL domain led to several unusual design choices, which we discuss in this section. For example, our expression language is very careful about choosing a representation for numerical values for which exact comparison is available—a property not enjoyed by IEEE 754 floating point values.

To explain the implementation of HSprial, we will use the Cooley-Tukey decimation in time and decimation in frequency formulas given in Theorem 2.1. The HSprial translations of these two decompositions are given in Listing 1. We have made use of Haskell’s support for Unicode operators to enable a direct mapping from mathematical formula to Haskell code. The differences are small—rather than representing the twiddle matrix $T_{rs}$ abstractly as an SPL term, we compute it with a function $\text{twid}$, which we do not show, and we have a special representation for permutations, which is why there is a $\text{Pi}$ data constructor wrapping the $\text{L}$ permutation.

As with much Haskell code, it is the type signature that needs the most explanation. The Cooley-Tukey rules take the two factors $r$ and $s$, which are integers, and a value that can have any type as long as this type is a member of the type class $\text{RootOfUnity}$, whose definition is shown in Listing 2. Recall that the Fourier transform is valid not just over the complex domain, but over a commutative ring with the necessary roots of unity. The operation that is fundamental to
cooleyTukeyDIT, cooleyTukeyDIF ::
  RootOfUnity a => Int -> Int -> a -> SPL a
cooleyTukeyDIT r s w =
  (F (w^s) ⊗ I s) × twid (r*s) s w ×
  (I r ⊗ F s (w^r)) × Pi (L (r*s) r)
cooleyTukeyDIF r s w =
  Pi (L (r*s) s) × (I r ⊗ F s (w^r)) ×
  twid (r*s) s w × (F r (w^s) ⊗ I s)

Listing 1. Haskell implementations of the DIT and DIF Cooley-Tukey rules.

class Fractional a => RootOfUnity a where
  rootOfUnity :: Int -> Int -> a
  omega :: Int -> a
  omega n = rootOfUnity n 1

instance RealFloat a => RootOfUnity (Complex a) where
  rootOfUnity n k = exp (-2*pi*i/fromIntegral n)^^k
  where
    i = 0:+1

Listing 2. The RootOfUnity type class and its instance for the type Complex a.

computing the discrete Fourier transform over a ring is that
of finding an n-th root of unity, captured in Haskell with
the RootOfUnity type class constraint in the type signature.
This allows our Cooley-Tukey rules to work for types other
than complex numbers, an ability we leverage in Section 4.3
to implement the number-theoretic transform (NTT).

SPL terms all have type SPL a for some index type a,
which specifies the type of the scalar elements on which
the SPL term operates. Since our eventual goal is to gener-ate
code for an SPL transform, we want to be able to write
transforms that operate not just on known constants, but
also on symbolic expressions that represent future input.
Compilation to C therefore requires an SPL formula of type
SPL (Exp a), where a value of type Exp a could be not only
a constant, but also a variable reference, array index, or other
(pure) mathematical operation. We now elaborate on HSpin-ral’s representation for expressions, SPL formulas, and the
other components that make up HSpiral.

3.1 Expression DSL

Parameterizing SPL formulas over expressions gives us great
flexibility in representing mathematical terms because it
allows us to compute transforms symbolically. Our represen-tation for expressions leverages Haskell’s support for
generalized algebraic data types (GADTs) [34] and is shown
in Listing 3. The expression language is standard, providing
support for the usual mathematical operations as well as
construction of complex values and projection of their real
and imaginary parts. Haskell’s type class mechanism allows
us to write natural code that constructs values of type Exp a
by implementing the appropriate instances of the standard
type numerical classes, e.g., Num, Fractional, and Floating.
The type class Typed seen in Listing 3 is instantiated only
for Haskell types that are also valid types in the embedded
expression and imperative languages. Using GADTs means
the type checker can refine type information when a pattern
match is performed. For example, when a value of type
Exp a is matched against the ImE data constructor, the type
checker refines its knowledge of the type a in Exp a, infer-
ing that it must have the form Complex b. GADTs allow
us to assign more accurate types to terms in our intermediate
languages while ensuring that the Haskell type checker
enforces type correct code.

What required more careful attention was the choice of
representation for constants. In particular, exact comparison
of complex values allows us to simplify more code through
optimizations like common subexpression elimination. Kise-
lyov and Taha [25] build knowledge of certain trigonometric
identities into their FFT code generator, but we go a step
further and utilize algebraic identities amongst rational sums
of roots of unity using cyclotomic polynomials to provide
additional optimizations [27]. The cyclotomic polynomial
Φn(x) is the minimal degree polynomial whose roots contain
the primitive n-roots of unity; computation in cyclotomic
fields Q(ω), rational sums of powers of roots of unity,
can be represented exactly as polynomials modulo cyclotomic
polynomials. Cyclotomic fields can be used to perform exact
computation with Gaussian rationals (numbers of the form
p + qi, where p and q are rational), square roots of rational
numbers, complex roots of unity, and sine and cosine of all
rational multiples of π.

While a representation that includes only roots of unity
is sufficient for exactly expressing many FFT formulas, in-
cluding both the decimation in time and the decimation in
frequency Cooley-Tukey formulas, we need the full power
of cyclotomic polynomials for the Rader [37] decomposition,
which involves sums of roots of unity, and the improved split-
radix decomposition we give in Section 4.2.2, which requires
summing terms involving sines and cosines of rational mul-
tiples of π. We use an existing Haskell library for cyclotomic

Listing 3. The Exp Haskell data type.
A Haskell Compiler for Signal Transforms

The representation for SPL terms is the simplest part of HSpi-
tions, we need to apply rewrite rules that match only

Our implementation is careful to preserve exact represen-
tations whenever possible, but the user of the expression
DSL does not have to worry about these details—he or she
need only write standard Haskell code. We conjecture that
our use of cyclotomic polynomials is what allows our code
generator to eliminate more common subexpressions, which
directly leads to the reduced number of multiplications re-
quired by our improved split-radix transform as compared
to previous work, as described in Section 5.2.

3.2 SPL Formulas

The representation for SPL terms is the simplest part of HSpi-
ral. Part of the declaration of the SPL data type is shown in
Listing 4. The E data constructor allows embedding a trans-
form represented as an explicit matrix in SPL—we describe
our support for matrices in Section 3.3. The identity matrix
is represented with the I data constructor, permutations can
be embedded in SPL using Pi, and diagonal matrices with
Diag. The Kronecker product, direct sum, and matrix prod-
uct operators have corresponding constructors as well. A
DFT of size \( n \) is represented with the DFT data constructor,
and a "rotated" DFT—a DFT parameterized by an arbitrary
root of unity—is represented with the F data constructor.

Besides a few smart constructors for forming tensor and
matrix products (the operators \( \otimes \) and \( \times \)) and direct
sums (the operator \( \oplus \)) that perform peephole optimizations
when constructing SPL terms, there is not much to the
SPL implementation—the real work is done within the
expression language (Section 3.1) and by the code genera-
tor (Section 3.4). Our design of the SPL data type attempts
to minimize the number of data constructors, preferring to
write function like

```haskell
data SPL a where
  E :: Matrix M e -> SPL e
  I :: Num e => Int -> SPL e
  Pi :: Permutation -> SPL e
  Diag :: V.Vector e -> SPL e
  Kron :: SPL e -> SPL e -> SPL e
  DSum :: SPL e -> SPL e -> SPL e
  Prod :: SPL e -> SPL e -> SPL e
  DFT :: RootOfUnity a => Int -> SPL a
  F :: RootOfUnity a => Int -> a -> SPL a
```

Listing 4. The SPL Haskell data type. Not all data construc-
tors are shown.

non-rotated DFTs, and in general there is not a constant-time
test that a root of unity is an \( n \)-th root of unity—elements of
\( \mathbb{Z}/p \) are one case where a constant-time test is unavailable.
Here and elsewhere, our choice of when to add a new data
constructor to a data type instead of writing a function that
builds a composite term is guided by the principle that a new
data constructor should only be added when doing so is
necessary to retain structure that we need to take advantage
of later. Here, we needed to retain knowledge that the root
of unity parametrizing the DFT was an \( n \)-th root of unity.

3.3 Array Computations

Our library for array computations builds on two tech-
niques that originally appeared in the Repa (REgular PARal-
lel arrays) Haskell library for high-performance array com-
putation: shape polymorphism [22] and the use of index
types to reflect an array’s underlying representation at the
type level [28]. While Repa’s goal is to support writing high-
performance Haskell code, our goal is to support the genera-
tion of high-performance code. Different requirements led
to a different library design.

The \( \text{toMatrix} \) function converts an SPL formula to an explicit
matrix representation, which is convenient for testing
and demonstrates several features of our array library.
Listing 5 shows the two cases that handle the DFT, corre-
sponding to the formula for constructing the DFT matrix:

\[
\text{DFT}_{n}(\omega) = \begin{bmatrix}
\omega^{ij}
\end{bmatrix}_{0 \leq i, j < n}
\]

One way to represent matrices in our library is as a
function from index to value. In Listing 5, the case
for the \( \text{F} \) data constructor uses this representation by
calling \( \text{fromFunction} \) with the array bounds and the
index-mapping function as arguments. Recall that a DFT
of size \( n \) parametrized by root of unity \( \omega \) is represented by
the SPL term \( \text{F} \ n \ s \ w \), so the first argument to \( \text{fromFunction} \)
uses the helper function \( \text{ix2} \) to construct the array bounds,
which is \( n \times n \). The second argument to \( \text{fromFunction} \)
is the locally-defined function \( f \), which maps indices to values.
It matches an array index using the pattern \( \text{Z} \cdot \cdot \text{i} \cdot \text{j} \) and
computes \( w^{*}(i\cdot j) \), i.e., \( \omega^{ij} \).

The \( \text{Z} \) data constructor is a zero-dimensional index, and
the \( \cdot \) data constructor adds one more dimension to an
existing index, so \( \text{Z} \cdot \cdot \text{i} \cdot \text{j} \) is a two-dimensional index. Both
\( \text{Z} \) and \( \cdot \) have identically-named \textit{type} constructors that

3https://hackage.haskell.org/package/cyclotomic
allow the number of dimensions to be checked by the compiler, so, for example, the compiler can statically enforce that matrix multiplication is only applied to two two-dimensional arrays. This support for shape polymorphism is borrowed directly from Repa [22].

The manifest function transforms the representation of an array without changing its values—it takes any array that supports indexing and returns an array where all values have been manifested and stored in memory. We can tell that toMatrix returns a manifest representation by its type signature since the representation type M is the first argument to the type constructor Matrix. The case for DFT n recursively calls toMatrix using the parameterized DFT. The omega function is a member of the RootOfUnity type class, and omega n computes the n-th root of unity.

Unlike Repa, our array library must fully support symbolic computation. Although one could construct Repa arrays that contain symbolic values of type Exp a, this is not enough to support our needs. For example, when generating code for a transform, we often need to generate loops whose index variables are used to compute array indexes. This means that our arrays may not only contain symbolic values, but that they must also support indexing using symbolic values. This requirement is what originally led us to write our own array library building on the work of Repa. Although we reuse Repa’s techniques for supporting shape polymorphism and representation polymorphism via indexed types, we could not re-use code from Repa due to our need to support symbolic computation.

A general array in HSpiral has type Array r sh e, where r is the type tag that signifies the array’s representation, sh is the shape of the array, e.g., two-dimensional, and e is the type of the elements contained in the array. Functions that work for any shape sh are shape polymorphic, and functions that work for any representation r are representation polymorphic. We have already seen the type Matrix M e in the type signature of toMatrix; Matrix is just a type alias for Array DIM2, so the type Matrix M e is equivalent to the type Array DIM2 M e. Most of the array representations available in our library are shown in Table 1 with their type tags. The function fromFunction constructs an array with the type tag D, that is, an array whose entries are given by a function that maps a statically known index to a value. For example, the local function f in Listing 5 maps an index of type Z:.Int:.Int to a value.

Like a delayed array (type tag D), a delayed symbolic array (type tag DS) is represented using a function from index to value; however, in the case of the delayed symbolic array, this function must accept a symbolic index. We could construct such an array with from$Function (from-symmetric-function) with a function whose first argument had type Z:.Exp Int:.Exp Int. This type tells us that the index-mapping function must be able to handle symbolic indices and cannot rely on receiving an index with known integer values.

Several of the representations were created based on the needs of the code generator. For example, the code generator often needs to work with array slices, which index into an underlying array at a starting offset and fixed stride, both of which may be symbolic. Rather than performing complex index manipulations in the code generator, we added the S representation, which allows the code generator to treat an array slice as it would any other array without worrying about index calculation. That is, if a function is representation polymorphic, we can use it unchanged with array slices since we have added a representation for slices that performs the needed index calculations automatically.

Another code transformation that is simplified by adding a new array representation instead of modifying the code generator is array scalarization [35], which replaces constant-indexed array references by scalar variables. The V representation allows array scalarization by representing an array as a list of symbolic expressions with the guarantee that each symbolic expression allows assignment. This is the key difference between the V and DS representations—the DS representation does not allow assignment. Representation polymorphic functions do not need to be rewritten to gain the benefits of array scalarization—they just need to be called with arguments that have the V representation.

Operations with side effects are expressed in the computation DSL, which we describe next. The CP array representation only allows computation—its underlying representation is a computation that can compute the array given a destination for its values. This representation is similar to push arrays, which appeared originally in Obsidian [6] and are now present in other Haskell array DSLs, including Feldspar [1]. The T representation is used during code generation to allow different code generation strategies to individually choose whether to delay computation by computing elements symbolically (representation DS) or to immediately compute an intermediate result (representation CP).

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>A delayed array (functional representation).</td>
</tr>
<tr>
<td>DS</td>
<td>A delayed symbolic array.</td>
</tr>
<tr>
<td>M</td>
<td>A manifest array (all values in memory).</td>
</tr>
<tr>
<td>S</td>
<td>An array consisting of a slice of an underlying array.</td>
</tr>
<tr>
<td>V</td>
<td>A virtual array whose elements are arbitrary expressions.</td>
</tr>
<tr>
<td>CP</td>
<td>An array that can be computed.</td>
</tr>
<tr>
<td>T</td>
<td>A transform array—either DS or CP.</td>
</tr>
</tbody>
</table>
Table 2. SPL formulas and corresponding pseudo-code. $T$ is the (square) matrix representation of the SPL term. The slice notation $[i:n:m]$ indicates an array slice starting at offset $i$ consisting of $m$ elements and having stride $n$. Subscripts give the dimensions of transforms, e.g., $I_n$ is the $n \times n$ identity matrix and $A_n$ is an arbitrary SPL formula whose matrix representation is of dimension $n \times n$.

<table>
<thead>
<tr>
<th>SPL</th>
<th>Pseudo-code for $y = Tx$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AB$</td>
<td>$t = B(x)$; $y = A(t)$;</td>
</tr>
<tr>
<td>$I_m \otimes A_n$</td>
<td>for ($i=0;i&lt;m;++i$) $y[i<em>m:n] = A(x[i</em>m:n])$;</td>
</tr>
<tr>
<td>$A_m \otimes I_n$</td>
<td>for ($i=0;i&lt;n;++i$) $y[i:m] = A(x[i:m])$;</td>
</tr>
<tr>
<td>$A_m \otimes B_n$</td>
<td>$y[0:1:m] = A(x[0:1:m])$; $y[m:1:n] = B(x[m:1:n])$;</td>
</tr>
</tbody>
</table>

### 3.4 Computation DSL

Once we have an SPL formula, we can convert it into a computation, which is represented using HSpiral’s computation DSL. The representation of computations is simple: a computation is a sequence of declarations and statements; each declaration defines a new mutable scalar, mutable array, or constant array; and each statement is either an assignment or a loop. Assignment re-uses the expression DSL, so the syntax of the computation DSL is very small—the computation DSL extends the pure expression DSL with imperative operations. Programmers don’t construct computation abstract syntax directly, but instead use a library of combinators. Most of these combinators operate in the DSL’s P “program” monad, which allows DSL users to generate code in a monadic style and reuse existing Haskell libraries for monadic programming. A computation in the P monad is run by calling runP, which returns a block of code, of type Block, consisting of a list of variable declarations and a list of statements.

Table 2 shows the pseudo-code corresponding to several SPL formulas; it mirrors the code template definitions given by Voronenko [42, p.20, Table 2.4] and Franchetti et al. [8]. Although SPL terms can be represented as matrices, we do not implement a SPL formula as a single matrix-vector multiplication. Instead, we take advantage of the structure of the SPL formula, which represents a sparse factorization of a transform, to implement transforms using fewer operations than a single matrix-vector product would require. This is why Table 2 shows the “matrix product” $AB$—really the product of two SPL sub-formulas $A$ and $B$—being implemented as the composition of two vector transformations.

The rules in Table 2 are implemented by the runSPL function in HSpiral, and the case for $I_m \otimes A_n$ is shown in Listing 6. This function takes two arguments: an SPL formula, of type SPL (Exp a) and an input vector that has the T representation, of type Vector T (Exp a), and it computes an output vector, also of type Vector T (Exp a). The fact that computation is being performed is captured in the return type of runSPL, which is P m (Vector T (Exp a)). The P type constructor is our program monad—actually a monad transformer [26] that adds code-generation capabilities to an underlying monad m, which must provide functionality specified by the MonadSpiral type class constraint.

The Haskell code for compiling $I_m \otimes A_n$ is a direct translation of the appropriate entry in Table 2. We first ensure that the argument vector x can be used as a data source by calling gather, which will compute x and allocate storage for it if it has not yet been computed. The idea is to be lazy about computing vectors, building computations that can compute an array when it is finally demanded, and only forcing this computation to occur when we really need the contents of the vector. This strategy has the benefit of automatically fusing many operations, such as multiple applications of permutations like I_n^T.

We construct the code generator that can compute the result of applying the operator to its argument with computeTransform, which takes as its argument a continuation that will assign the result of the transformation to the destination given as parameter y. This function does not immediately compute the result, but constructs a code generator of type P (Vector T (Exp a)). The code generator will be called when the final destination for the result is known, as in the recursive call to runSPL with argument a in line 10 of Listing 6.

The continuation uses two combinators to generate code. The first is forP, which constructs a for loop, and the second is the operator (.<-.), which performs array assignment. There are two uses of the slice function, corresponding to the two slices in Table 2. We also make a recursive call to runSPL to run the transform $A_n$ on a slice of the computed argument, t. The fromGather function takes a
Vector \( r \) (Exp a) and delays its computation, converting it to a Vector T (Exp a).

3.4.1 Optimizing Computations

Writing a code generator in this style allows for several optimizations to happen “under the covers.” The first optimization we perform is selective loop unrolling. The decision of when to unroll a loop is made automatically by the forP combinator. Although the programmer using forP can leave unrolling decisions to the combinator, we do provide additional functions for explicit control of unrolling. Currently, forP uses the loop bounds to decide when to unroll a loop; however, because it has access to the code generated for the body of the loop by its continuation, it could use information about the complexity of the loop body to make unrolling decisions. We leave such an analysis to future work.

A second optimization performed behind the scenes in the P monad is common subexpression elimination (CSE), which is performed whenever code is generated for an assignment. For example, if \( x \) comes before \( y \) in the chosen order, then \( y - x \) is rewritten to \( -(x - y) \).

- Every negative constant is rewritten to the negation of a positive constant.
- A canonical order is chosen for variables when they occur as sub-terms of a binary operator. For example, if \( x \) comes before \( y \) in the chosen order, then \( y - x \) is rewritten to \( -(x - y) \).
- For a constant \( k, e * k \) is rewritten to \( k * e \).
- Negation is normalized, e.g., \( e_1 + (-e_2) \) is rewritten to \( e_1 - e_2 \), and \( e_1 - (-e_2) \) is rewritten to \( e_1 + e_2 \).

The first two rewrites are also performed by Frigo [14], as Frigo noted, constants that arise in the FFT often come in pairs \( k \) and \(-k\), and the first rewrite can lead to a reduced operation count. In particular, this rewrite allows us to automatically take advantage of the structure of the odd eight roots of unity, which have the form \( (\pm 1 \pm i)/\sqrt{2} \), to save a multiplication.

3.5 Compilation to C

All optimizations to SPL programs are performed at the level of the computation DSL. The C back end straightforwardly translates the computation DSL’s AST into C using the language-c-quote\(^4\) library for C quasi quoting [29]. Although we currently have only one back end, because most optimizations are done using the computation language, additional back ends would be easy to add.

3.6 Performing Search

The final component of HSpiral is search. Although HSpiral’s search mechanism is general, we have currently only implemented search over operation count, i.e., we can search for optimizations.

---

\(^4\)https://github.com/mainland/language-c-quote

Listing 7. Cooley-Tukey breakdown rule in HSpiral.

```hs
cooleyTukeyBreakdown (F n w) =
msum [return $ cooleyTukeyDIF r s w | (r, s) <- fs] <|>
msum [return $ cooleyTukeyDIT r s w | (r, s) <- fs]
where
  fs :: [(Int, Int)]
  fs = factors n
cooleyTukeyBreakdown _ = mzero
```

coolesyTukeyBreakdown rewrites of SPL formulas, like DFT 128, representing the 128-point DFT, that minimize operation count. Our implementation is built on a backtracking monad transformer [19, 23]. During search, SPL formulas are rewritten using an extensible set of user-supplied rules. Listing 7 shows one rule that decomposes a DFT into smaller transforms using the Cooley-Tukey DIF and DIT factorizations. This rule uses standard operations on monads that are members of MonadPlus, including mzero to represent failure (backtracking), (<>|>) for choice, and msum for choice amongst members of a list. Rules can be composed with (<>|>) or msum similarly to their use in cooleyTukeyBreakdown.

3.7 Summary

Although tailored to implementing a compiler for SPL, many of HSpiral’s components would be individually useful in other settings—even the computation DSL and the C back end are re-usable. Furthermore, by using our library, programmers get many optimizations “for free.” For example, loop unrolling and CSE are performed in the P monad automatically, and other optimizations, like constant folding, are performed during expression construction by the type class instances defined for the expression DSL. Our library makes code generation for SPL operations fairly straightforward. Part of the reason behind this ease of use is our abstraction of different array representations into a separate library for array computation, which is also independently useful.

4 Case Studies

In this section, we present three case studies illustrating the use of SPL to derive, represent, and implement FFT algorithms. The first case study is Rader’s algorithm for computing the DFT when the number of inputs is prime and hence the Cooley-Tukey algorithm is not applicable. The second case study involves the split-radix algorithm [46] and an improvement due to Johnson and Frigo [21], which reduces the number of real multiplications used by the Cooley-Tukey algorithm when the input data is complex. The third case study is the so-called number-theoretic transform (NTT), which computes the DFT over the integers modulo a prime \( p \). These examples illustrate the power and ease of use of HSpiral. Rader’s algorithm [37] requires the generalized matrix

---

Geoffrey Mainland and Jeremy Johnson

GPCE’17, October 23–24, 2017, Vancouver, Canada
constructors, and our implementation shows how the mathematical description can be easily translated into HSpiral. The split-radix algorithm and its improvement illustrate how to encode a complicated algorithm and use HSpiral to easily explore its performance, enabling us to not only implement the new algorithm, but also to obtain a further reduction in operation count using HSpiral’s code optimization. The implementation of the NTT leverages HSpiral’s use of Haskell’s type system to add support for integers modulo a prime p.

4.1 Rader’s Algorithm

Rader’s algorithm provides an efficient way of computing DFT_p for prime p. Its derivation depends on the primitive element theorem, which states that all non-zero elements of \( \mathbb{Z}/p \) can be written as a power of some generator \( a \in \mathbb{Z}/p \). The Rader decomposition is given by the following decomposition

\[
D_p = \begin{bmatrix}
1 & 1 \\
1 & \delta_1 \\
\vdots & \vdots \\
\delta_{p-1} & \delta_{p-1}
\end{bmatrix}
\]

where \( \delta_k = \frac{1}{DFT_{p-1}(\nu)} \) and \( \nu \) is the \( p-1 \)-st root of unity, and \( D_p \) is given by

\[
D = \begin{bmatrix}
\omega & \cdots \\
\cdots & \cdots \\
\omega^{p-1} & \cdots
\end{bmatrix}
\]

The HSpiral implementation of this decomposition, shown in Listing 8, demonstrates the advantages of embedding SPL in Haskell—the implementation uses Haskell extensively to construct the Rader decomposition. For example, the vector \( \delta \) uses our array computation library to explicitly compute a matrix-vector product via \( \text{mv} \). The rader function also uses number-theoretic Haskell functions to find a generator for \( \mathbb{Z}/p \) (gen), find the inverse of an element in \( \mathbb{Z}/p \) (inv), and perform modular exponentiation efficiently (modExp). Furthermore, it leverages idiomatic language features like list comprehensions. If we had built SPL as a stand-alone DSL, we would have had to implement all of these features ourselves.

The Rader decomposition also demonstrates the advantage of using cyclotomic polynomials. Using the cyclotomic identity \( \Phi_n(\omega) = 0 \), we infer that \( \delta_1 = 1/(p-1) \), which is detected by our simplification rules. This reduces the operation count when the input data is complex, because we know that the imaginary component of \( \delta_1 \) is exactly zero—without cyclotomic polynomials, \( \delta_1 \) is computed as a complex number with a very small, though non-zero, imaginary component. Additional cyclotomic identities detect that \( \delta_{(p-1)/2+1} \) is either real or purely imaginary, further reducing the operation count.

4.2 Split-Radix Algorithm

The split-radix algorithm [46] reduces the number of real multiplications used by the Cooley-Tukey algorithm by taking advantage of the fact that a multiplication by \( i = \sqrt{-1} \) does not require a real multiplication. It is possible to derive an algorithm equivalent to the split-radix algorithm using common subexpression elimination on the code from the generated radix two Cooley-Tukey algorithm [25]. However, it can be formally derived in SPL from the Cooley-Tukey rule using properties of roots of unity and matrix identities. We give the result of this derivation here.

**Theorem 4.1 (Split-Radix Rule).**

\[
\text{DFT}_{4p} = (G_4 \otimes I_p)(I_{2p} \oplus W_p(\omega) \oplus W_p(\omega^3))
\]

where \( G_4 = (\text{DFT}_{2p} \oplus \text{DFT}_p)(I_{2p} \oplus L_{2}^{4p})I_{2}^{4p} \)

4.2.1 Split-Radix through Code Optimization

The split-radix algorithm reduces real multiplications by combining twiddle factors from two stages and precomputing the products (constant folding) at the cost of multiplications by \( i \)—the saving would not result if multiplication by \( i \) involved real multiplications. In order to combine constants, it is necessary to rearrange operations, which is enabled by factoring \( i \) from roots of unity. This suggests that the split-radix algorithm could be derived using code optimization
rules. Indeed, the rules that capture the derivation of the split-radix algorithm are

1. \( \omega^k t_k + \omega^{k+2} t_{k+p} \rightarrow \omega^k t_k + \omega^{k+2} t_{k+p} \)
2. \( \omega^{k+2} t_{k} - \omega^{k+2} t_{k+p} \rightarrow \omega^p (\omega^k t_k - \omega^{k+2} t_{k+p}) \)

However, when applied naturally (from inputs to outputs) to the decimation in time FFT, these rules will be applied to smaller transforms first, which prevents them from being applied to larger transforms where greater savings arise. The result fails to achieve the full savings of the split-radix algorithm. To remedy this, the computation graph can be transposed, as done by FFTW [14]. Such transposition is valid because the DFT is symmetric.

The transposed graph corresponds to the decimation in frequency FFT, which was used by Kiselev and Taha [25] to derive split-radix operation counts using the dual optimization to those presented above. The dual optimization rule is

\[ \omega^k t_k \pm \omega^{k+2} t_{k+p} \rightarrow \omega^k (t_k \pm \omega^p t_{k+p}) \]

where \( \omega^p \) is a 4-th root of unity. Using decimation in frequency moves the larger twiddle factors towards the inputs and obtains an opcunt reduction equivalent to the split-radix algorithm starting with the radix two Cooley-Tukey algorithm using this optimization rule. The rules as presented would be stated in a more general way, e.g.,

\[ \omega_1 t_1 \pm \omega_2 t_2 \rightarrow \omega_1 (t_1 \pm \omega_2 / \omega_1 t_2) \]

when \( \omega_1 \) or \( \omega_2 / \omega_1 \) is a 4-th root of unity; however, we wanted to illustrate where the savings is coming from and why decimation in frequency is preferable to decimation in time when using such optimizations. When we implement this rewrite rule in our system, we obtain the same opcounts for the decimation in frequency transform as for split-radix.

### 4.2.2 Improved Split-Radix Algorithm

We also derive and implement the improved version of the split-radix algorithm, due to Johnson and Frigo [21], that further reduces the number of real multiplications. The improved algorithm scales the complex twiddle factors \( \omega_N^k = \exp(2\pi k/N) = \cos(2\pi k/N) + i\sin(2\pi k/N) \) by \( 1/\cos(2\pi k/N) \) so that the scaled twiddle factor \( 1 + \tan(2\pi k/N) \) can be computed using two real multiplications. The extra multiplications required by the corresponding scaling of the input data are cleverly incorporated into the recursive calls. This leads to four mutually recursive algorithms with different scaling factors, which can be easily expressed in SPL. Our implementation of the resulting improved split-radix formulas obtained further optimizations and reduced the operation counts obtained by Johnson and Frigo.

The derivation of the improved split-radix algorithm begins with a modification of the split-radix rule called the conjugate pair split-radix algorithm.

**Theorem 4.2** (Conjugate Pair Split-Radix Rule).

\[
\text{DFT}_{4p} = (G_4 \otimes I_p)(I_{2p} \oplus W_p(\omega) \oplus W_p(\omega^{-1}))
\]

\[
= (DFT_{2p} \oplus DFT_p \oplus DFT_{2p}) (I_{2p} \oplus L_{2p}^p) L_{2p}^4
\]

where \( G_4 = (DFT_2 \otimes I_2) T_2^4 (I_2 \oplus DFT_2) \) and \( S_p \) is the shift matrix, which cyclically rotates its input.

The key difference from the split-radix algorithm is the twiddle factor matrix \( I_{2p} \oplus W_p(\omega) \oplus W_p(\omega^{-1}) \) compared to \( I_{2p} \oplus W_p(\omega) \oplus W_p(\omega^3) \), which has diagonal elements that come in conjugate pairs. The benefit of this is that the same scale factor can be used for both \( \omega^k \) and \( \omega^{-k} \). This modification follows immediately from the convolution theorem since \( DFT_p(\omega^4) = DFT_p(\omega^4) S_p^{-1} S_p = W_p(\omega^4) DFT_p(\omega^4) S_p \) and \( W_p(\omega^3) W_p(\omega^{-4}) = W_p(\omega^{-1}) \).

The rules that capture the improved split-radix algorithm introduce scaling factors into the twiddle factor matrices along with the corresponding inverse scale factors in the recursive calls. Let \( T_s = \text{Diag}(s_{N,k}, k = 0, \ldots, p - 1) \) and \( F_{sp} = \text{Diag}(1/s_{N,k}, k = 0, \ldots, p - 1) F_{Sp} \). The general scaled split-radix rule is

\[
\text{DFT}_{4p} = (G_4 \otimes I_p)(I_{2p} \oplus W_p(\omega) T_s \oplus W_p(\omega^{-1})) T_s
\]

\[
= (DFT_{2p} \oplus F_{sp} \oplus F_{sp} S_p)(I_{2p} \oplus L_{2p}^p) L_{2p}^4
\]

The particular scale factors used are defined by

\[
s_{N,k} = \begin{cases} 1 & N \leq 4 \\ S_{N/4,k} \cos\left(\frac{2\pi k}{N}\right) & k_4 \leq N/8 \\ S_{N/4,k} \sin\left(\frac{2\pi k}{N}\right) & k_4 > N/8 \end{cases}
\]

where \( k = k \mod (N/4) \). These numbers are \( N/4 \) periodic, i.e. \( s_{N,k+N/4} = s_{N,k} \), which is needed for the derivation of the algorithm as this will allow the scale factors to commute past various butterfly stages.

Applying the scaled split-radix rule with scale factors \( s_{N,k}, k = 0, \ldots, p - 1 \) combined with the twiddle factors leads to the improved split-radix rules.

**Theorem 4.3** (Improved Split-Radix Rule).

1. **[Improved Split-Radix Rule 1—unscaled DFT]**

\[
\text{DFT}_{4p} = (G_4 \otimes I_p)(I_{2p} + T_s p \otimes T_s^*)
\]

\[
= (DFT_{2p} \oplus F_{sp} \oplus F_{sp} S_p)(I_{2p} \oplus L_{2p}^p) L_{2p}^4
\]

where \( T_{sp} = \text{Diag}(\omega^k s_{p,k}, k = 0, \ldots, p - 1) \)

\( T_{sp}^* = \text{Diag}(\omega^{-k} s_{p,k}, k = 0, \ldots, p - 1) \)

\( F_{sp} = \text{Diag}(1/s_{p,k}, k = 0, \ldots, p - 1) F_{sp} \)

2. **[Improved Split-Radix Rule 2—scaled DFT]**

\[
\text{Fs}_{4p} = (G_4 \otimes I_p)(I_{2p} \oplus T_{sp} p \otimes T_{sp}^*)
\]

\[
= (F_{sp} S_{2p} \oplus F_{sp} S_{2p} S_p)(I_{2p} \oplus L_{2p}^p) L_{2p}^4
\]

\[
= (F_{sp} S_{sp} S_{2p} \oplus F_{sp} S_{sp} S_p S_p)(I_{2p} \oplus L_{2p}^p) L_{2p}^4
\]
The final case study we present is an attempt to add support for the Number-Theoretic Transform (NTT), which is another name for the DFT over the ring $\mathbb{Z}/p$. Our implementation leverages an existing Haskell library\(^\dagger\) that supports computation with values in $\mathbb{Z}/p$. When compiling to C, we represent values of type $\mathbb{Z}/p$ as 64-bit integers. We could add support for larger primes by using GMP.

Adding support for the NTT did not require us to modify the SPL language—our choice of representation for DFTs in SPL guarantees that they will work for any type that is an instance of `RootOfUnity`. Writing an instance for the type $\mathbb{Z}/p$ requires only a few lines of code. Although we did not need to modify the SPL language, we did need to modify the expression DSL and the code generator to support values of type $\mathbb{Z}/p$.

Our modifications to the expression DSL totaled approximately 50 lines of code, including type class instances. The C code generator had to be modified to handle modular values, which required an additional 12 lines of code. In all, the changes required to support $\mathbb{Z}/p$ totaled around 100 lines of code, not including tests. A great deal of effort—a whole PhD’s worth [31]—went into adding support for modular transforms to SPL. In fairness, we do not support nearly all the features of Meng [31], but our efforts demonstrate that HSpiral is easily extensible. Haskell’s type system was critical to our efforts, because it allowed us to easily abstract over types and make many operations type-directed. For example, Meng had to copy the code for each FFT decomposition he wanted to use for $\mathbb{Z}/p$, whereas our FFT decompositions are written so that they work for any commutative ring with the necessary roots of unity, i.e., any type for which we can define a `RootOfUnity` instance.

\(^\dagger\)https://hackage.haskell.org/package/modular-arithmetic

4.3 The Number-Theoretic Transform

The final case study we present is an attempt to add support for the Number-Theoretic Transform (NTT), which is another name for the DFT over the ring $\mathbb{Z}/p$. Our implementation leverages an existing Haskell library\(^\dagger\) that supports computation with values in $\mathbb{Z}/p$. When compiling to C, we represent values of type $\mathbb{Z}/p$ as 64-bit integers. We could add support for larger primes by using GMP.

Adding support for the NTT did not require us to modify the SPL language—our choice of representation for DFTs in SPL guarantees that they will work for any type that is an instance of `RootOfUnity`. Writing an instance for the type $\mathbb{Z}/p$ requires only a few lines of code. Although we did not need to modify the SPL language, we did need to modify the expression DSL and the code generator to support values of type $\mathbb{Z}/p$.

Our modifications to the expression DSL totaled approximately 50 lines of code, including type class instances. The C code generator had to be modified to handle modular values, which required an additional 12 lines of code. In all, the changes required to support $\mathbb{Z}/p$ totaled around 100 lines of code, not including tests. A great deal of effort—a whole PhD’s worth [31]—went into adding support for modular transforms to SPL. In fairness, we do not support nearly all the features of Meng [31], but our efforts demonstrate that HSpiral is easily extensible. Haskell’s type system was critical to our efforts, because it allowed us to easily abstract over types and make many operations type-directed. For example, Meng had to copy the code for each FFT decomposition he wanted to use for $\mathbb{Z}/p$, whereas our FFT decompositions are written so that they work for any commutative ring with the necessary roots of unity, i.e., any type for which we can define a `RootOfUnity` instance.

5 Evaluation

5.1 Execution Time

The performance of FFT implementations generated by HSpiral is compared to implementations generated by Spiral and FFTW3 3.3.4 in Figure 1. The Spiral implementations were taken from the Spiral web site. Because HSpiral does not yet support vector instructions, we compared un-vectorized implementations from Spiral and FFTW3. Data was collected on an i7–4770 CPU running at 3.40GHz under Ubuntu 16.04 (x64), generated C code was compiled with GCC 5.4, and all runs were repeated 100 times. Following Xiong et al. [45], we measure performance in pseudo-flops/cycle, computed as $5n \log_2(n)$ for a DFT of size $n$, since this is an upper bound on the number of arithmetic operations needed to compute the FFT.

\(^\dagger\)http://spiral.ece.cmu.edu/fft_scalar/
\(^\dagger\)\(^\dagger\)-march=native -mtune=native -Ofast

Figure 1. Performance of HSpiral code relative to Spiral and FFTW 3.3.4. Pseudo-flops are calculated as $5n \log_2(n)$.
Table 3. Operation counts for split-radix implementations. Columns other than “Johnson and Frigo” are computed by our implementation. Our count for split-radix matches Heideman and Burrus [18].

<table>
<thead>
<tr>
<th>Size</th>
<th>Split-radix</th>
<th>Johnson and Frigo [21]</th>
<th>Improved</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>168</td>
<td>—</td>
<td>168</td>
</tr>
<tr>
<td>32</td>
<td>456</td>
<td>—</td>
<td>448</td>
</tr>
<tr>
<td>64</td>
<td>1160</td>
<td>1152</td>
<td>1128</td>
</tr>
<tr>
<td>128</td>
<td>2824</td>
<td>2792</td>
<td>2744</td>
</tr>
<tr>
<td>256</td>
<td>6664</td>
<td>6552</td>
<td>6464</td>
</tr>
<tr>
<td>512</td>
<td>15368</td>
<td>15048</td>
<td>14848</td>
</tr>
<tr>
<td>1024</td>
<td>34824</td>
<td>33968</td>
<td>33544</td>
</tr>
</tbody>
</table>

Since our current search strategy seeks to minimize opcounts, it results in implementations that consist of straight-line code. This strategy works for small FFT sizes, but fails to perform well when $n$ is larger, as Figure 1 shows. Since our primary goal with HSpiral was to design a system that is good for experimenting with new digital signal processing (DSP) transforms, we are happy that performance is good for small $n$. We expect that altering our search strategy will lead to faster implementations. Although opcount-minimization is not the best strategy for modern machines with large caches, it is a good strategy for other platforms, like FPGAs.

5.2 Operation Counts for Our Improved Split-Radix
Operation counts for our improved split-radix algorithm (Section 4.2.2) are given in Table 3. Our implementation improves on the results reported by Johnson and Frigo [21]. We are not sure where this savings comes from. One possibility is that the decomposition in Theorem 4.3 absorbs some factors that are not absorbed by the decomposition given by Johnson and Frigo. We suspect that our use of cyclotomic polynomials led to the additional savings by identifying more equal constant factors, allowing CSE to eliminate more multiplications.

6 Related Work
Our work builds directly on SPL [45] and Spiral [35]. Much of the related work in this area was described in Section 1. Spiral is implemented in GAP [39], a computer algebra system without support for strong types. The most recently publicly-available SPL implementation is from 2002 [44], making it difficult for other researchers to leverage Spiral’s advances.

Ofenbeck et al. [33] implement a subset of SPIRAL in Scala, much as we implement SPIRAL in Haskell. Although our implementation is roughly equivalent in functionality to theirs, we also provide a reusable search mechanism with compositional search rules. Our expression and computation DSLs stand in for the Scala LMS framework [38], but our array library does not seem to have a Scala equivalent.

Kiselyov and Taha [25] build on their own prior work [24] using MetaOCaml [3] to implement FFT kernels. By careful choice of abstract domain, they obtain opcounts equal to the split-radix transform from the decimation in frequency transform. We show in Section 4.2.1 that adding a rewrite rule to our system yields the same opcounts for the DIT decomposition and explain why this is the case.

FFTW’s [14] transform generator is implemented in OCaml. Its optimizer is written in an explicit monadic style, making for somewhat awkward code. Haskell’s built-in support for do notation is a better fit for monadic code. FFTW’s DAG-reversal trick in effect leverages the same rewrite rule used by Kiselyov and Taha.

There are many Haskell DSLs that bear some similarity to HSpiral, including the GPU DSLs Obsidian [6], Nikola [30], and Accelerate[4], and the DSP DSL Feldspar [1]. Repa [22, 28] pioneered the technique of using a type index to reflect information about data representation in a term’s type.

Our improved split-radix formula and code generator improves on the opcounts reported by Johnson and Frigo [21], which for a number of years were the smallest known operation counts for FFTs of size $2^n$. The currently-known lowest operation count for FFTs of size $2^n$ was obtained by Zheng et al. [47].

7 Conclusions and Future Work
Haskell provides an excellent foundation on which to build a code generator. In particular, we leveraged the following Haskell language features to build HSpiral:

1. Type classes allowed us to manipulate symbolic expressions using standard Haskell functions.
2. Monads provided the infrastructure needed to embed a code generation DSL for monadic computations in Haskell.
3. GADTs allowed us to index expressions and SPL formulas with their type while propagating information about these type indexes to the type checker when pattern matching on terms.
4. Type families and index types [5] enable the type-indexed approach we use to implement our Repa-inspired array library.

As well as implementing a reusable framework for code generation, we expressed the modified split-radix decomposition of Johnson and Frigo [21] in SPL, implemented it in our framework, and generated code with fewer operations than that produced by prior work. We also explained prior results that used rewrite rules to generate split-radix opcounts from the decimation in frequency DFT [14, 25].

We are working to extend HSpiral to support vectorization [10, 13] and multicore [12]. We also plan to add CUDA and VHDL back-ends. We hope that our system will be useful to other researchers, and we plan to use it to explore new DSP transforms.
A Haskell Compiler for Signal Transforms

References


