Spiral in Scala: Towards the Systematic Construction of Generators for Performance Libraries

Georg Ofenbeck[†] Tiark Rompf [*‡] Alen Stojanov[†] Martin Odersky[*] Markus Püschel[†]

[†]Dept. of Computer Science, ETH Zurich: {ofgeorg, astojanov, pueschel}@inf.ethz.ch
[‡]Oracle Labs: {first.last}@oracle.com [*]EPFL: {first.last}@epfl.ch

Abstract

Program generators for high performance libraries are an appealing solution to the recurring problem of porting and optimizing code with every new processor generation, but only few such generators exist to date. This is due to not only the difficulty of the design, but also of the actual implementation, which often results in an ad-hoc collection of standalone programs and scripts that are hard to extend, maintain, or reuse. In this paper we ask whether and which programming language concepts and features are needed to enable a more systematic construction of such generators. The systematic approach we advocate extrapolates from existing generators: a) describing the problem and algorithmic knowledge using one, or several, domain-specific languages (DSLs), b) expressing optimizations and choices as rewrite rules on DSL programs, c) designing data structures that can be configured to control the type of code that is generated and the data representation used, and d) using autotuning to select the best-performing alternative. As a case study, we implement a small, but representative subset of Spiral in Scala using the Lightweight Modular Staging (LMS) framework. The first main contribution of this paper is the realization of c) using type classes to abstract over staging decisions, i.e. which pieces of a computation are performed immediately and for which pieces code is generated. Specifically, we abstract over different complex data representations jointly with different code representations including generating loops versus unrolled code with scalar replacement—a crucial and usually tedious performance transformation. The second main contribution is to provide full support for a) and d) within the LMS framework: we extend LMS to support translation between different DSLs and autotuning through search.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program synthesis, Program transformation; D.3.3 [*Programming Languages*]: Language Constructs and Features – Abstract data types; D.3.4 [*Programming Languages*]: Processors – Code generation, Optimization, Run-time environments

Keywords Synthesis, Abstraction over Staging, Selective Precomputation, Scalar Replacement, Data Representation

1. Introduction

The development of highest performance code on modern processors is extremely difficult due to deep memory hierarchies, vector instructions, multiple cores, and inherent limitations of compilers. The problem is particularly noticeable for library functions of mathematical nature (e.g., BLAS, FFT, filters, Viterbi decoders) that are performance-critical in areas such as multimedia processing, computer vision, graphics, machine learning, or scientific computing. Experience shows that a straightforward implementation often underperforms by one or two orders of magnitude compared to highly tuned code. The latter is often highly specialized to a platform which makes porting very costly (e.g., Intel's IPP library includes different FFT code, likely written in assembly, for Pentium, Core, Itanium, and Atom).

One appealing solution to the problem of optimizing and porting libraries are program generators that automatically produce highest performance libraries for a given platform from a high level description. When the platform is upgraded, the code is regenerated, possibly after an extension of the generator if new features need to be supported (e.g., longer vectors in the architecture as in AVX versus SSE). Building such a generator is difficult, which is the reason that only very few exist to date. The difficulty comes from both the problem of designing an extensible approach to perform all the optimizations the compiler is unable to do and the actual implementation of the generator. The latter often results in an ad-hoc collection of stand-alone programs or scripts. These get one particular job done but are hard to extend, reuse, or further develop, which is a major impediment to progress.

We believe that a programming environment that provides suitable advanced programming concepts should offer a solution to this problem. Hence, the motivating question for this paper is: *Which tools and features provided by programming languages and environments can facilitate the development of generators for performance libraries?* First, we inspect existing generators to derive a common, systematic approach. Then we show with a case study how the components of this approach can be realized using high-level language features and programming techniques.

Program generators for performance. A few program generators have been built for mathematical functionality with highest performance as objective. Examples include the FFTW codelet generator (codegen) for small transforms [15], ATLAS [41], Eigen [1], and Build to Order BLAS [4] for basic linear algebra functions, Spiral for linear transforms [26], the OSKI kernel generator for sparse linear algebra [39], FLAME for linear algebra [16], cvxgen for optimization problems [24], and FEniCS for finite element methods [2]. In most cases, the starting point is a description in a domain-specific language (DSL); where it is not (e.g., ATLAS, which only uses parameters) porting to new platform features (e.g., vectorization) or functions is difficult. In many cases, the DSL is used only to specify the input (e.g., in cvxgen, FEniCS), in some cases to also represent the algorithm (e.g., Flame, Spiral), and sometimes also to perform optimizations through DSL rewriting (e.g., Spiral). Some generators use search over alternatives to tune (e.g., ATLAS, OSKI) some do not (e.g., FFTW codegen, OSKI kernel generator). Several performance optimizations are relevant for most domains (e.g., loop unrolling combined with scalar replacement [5], precomputation, and specialization).

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 ACM 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.

GPCE '13, October 27–28, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2373-4/13/10. . . $15.00.
http://dx.doi.org/10.1145/2517208.2517228

These generators have been implemented in a large variety of environments. Some are built from scratch (e.g., ATLAS, cvxgen), others make use of a particular programming environment: e.g., OCaml (FFTW codegen), Mathematica (parts of FLAME), the computer algebra system GAP (Spiral). UFL in FEniCS is a standalone languages. Eigen is a collection of C++ templates that perform optimizations during preprocessing.

Systematic construction of program generators. Extrapolating from the commonalities of all these generators, we propose the following systematic approach to construct program generator implementations. Instantiating this approach in a problem domain (such as the ones above) is an orthogonal research question.

- *Describe problem and algorithmic knowledge through one or multiple levels of DSLs.* Program generators need to model problems and algorithms at a high level of abstraction and may need to optimize code at multiple intermediate abstraction levels. For example, FFTW codegen's input is a sum representation of FFTs but most optimizations are done on DAGs. Spiral uses three internal DSLs and rewriting for loop optimizations and parallelization. Successively lower-level DSLs are a natural choice to express these various stages of program generation.
- *Specify certain optimizations and algorithmic choices as rewrite rules on DSL programs.* DSLs can be used for high-level optimization through rewriting (e.g., parallelization in Spiral) but rewrite rules can also be used to express algorithmic choices. Doing so facilitates empirical search ("autotuning"), which in many cases is required to achieve optimal performance.
- *Design high-level data structures that can be parametrized to generate multiple low-level representations.* Often generated libraries need to support multiple input/output data formats. A common example is interleaved or split or C99 format for complex vectors, meaning there will be one library function per format. A generator should be able to abstract over this choice of low-level data formats to ensure maximal compatibility [3].
- *Rely on common infrastructure for recurring low-level transformations.* There are certain transformations common in high performance code that are necessary but particularly unpleasant to implement and maintain manually. Examples include a) unrolling with scalar replacement, b) selective precomputation during code production or initialization, and c) specialization (e.g., to a partially known input). Since these transformations are so common, they should be implemented in a portable way using suitable language features.

While the first two points, DSLs and rewriting, are well-studied topics and supported by existing tools, the latter two points have, to our best knowledge, only been realized in ad-hoc ways. It is a main contribution of this paper to demonstrate how all four points, including the last two, can be achieved with the help of high-level programming language features. To do so we utilize a case study implemented in a specific environment, which we describe in more detail in the following section. However, we emphasize that any programming environment that offers the needed language features can be used instead.

Language support for program generation. For already quite some time, the programming languages community has proposed multi-stage programming using quasi-quotation as a means to make program generation more principled and tractable [34]. However, most approaches remained a thin layer over syntactic composition of program fragments and did not offer facilities beyond serving as a type safe assembly language for program generation. We provide a more detailed discussion of related work in Section 5.

The recently introduced LMS [28, 30] framework works at a higher level than pure composition of code fragments; it is a library-based staging approach that offers an extensible compiler framework with a rich set of features, including transformations on an intermediate representation and different code generators. LMS has already been applied successfully to implement a range of performance-oriented, high-level domain specific languages (DSLs) in the Delite framework [7, 9, 32]; however, the requirements for generators of highest performance libraries go considerably beyond the use of LMS to date. First and foremost, previous LMS DSLs were designed to be user-facing, not as internal languages for program generators. Thus, no particular support for parameterizing DSL programs over low-level generation choices was available. While LMS has been equipped with program transformation support within a single intermediate representation [31], there was no support for translating between different DSLs. Furthermore, LMS did not provide support for autotuning and had only been used to generate moderately large pieces of code. Consequently, generating code as large as several MB caused serious performance problems which had to be addressed. Finally, while LMS has always used types to denote staged expressions, programming techniques that *abstract over* whether a certain expression is staged had not been studied.

Contributions. In summary, this paper makes the following contributions:

- We conduct a case study for the systematic construction of a program generator in the sense outlined before: the implementation of a subset of Spiral and FFTW codegen inside Scala with LMS. The subset covers the generation of fixed input size C code for FFTs as explained in [12, 15, 27, 42]. It does not cover transforms other than the FFT (Spiral covers more than 30), or the generation of vectorized or parallel code as explained in [13, 14]. However, even though the latter extensions are substantial, they are all based on rewriting. Only the generation of general input size libraries as described in [38] requires new techniques and is subject of future research.
- In implementing this case study, we develop novel programming techniques to address the challenges of parameterizing data structures over code generation choices and implementing transformations like unrolling with scalar replacement. Specifically, we show that with LMS, the type class pattern [40] is a natural fit to abstract over staging decisions, i.e., which pieces of a computation are performed immediately and for which pieces code is generated. More importantly, we show that this mechanism can be applied to data structures to decide which parts of a nested data structure are staged and which only exist at code generation time. This enables us to use a single generator pipeline that abstracts over all required data layouts. A particular layout can be chosen by instantiating the pipeline with the proper types. Coupled with selective staging of loops, this directly leads to an arguably elegant and modular implementation of various loop unrolling and scalar replacement schemes.
- We pushed the LMS framework beyond what was done previously. Novel are in particular the translation between different DSLs (which are not user-facing, but internal steps in the program generation pipeline), empirical autotuning through search, and performance optimizations inside the LMS framework to support the generation of much larger programs.

The source code accompanying this paper is available at spiral.net.

2. Background

We provide necessary background on Spiral and on the LMS framework [28, 30] in Scala.

2.1 Spiral

Spiral is a library generator for linear transforms such as the discrete Fourier transform (DFT). The version we consider here gener-

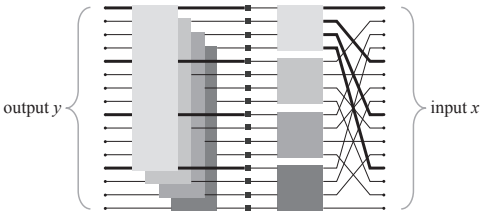

Figure 1. FFT (1) dataflow (right to left) for $n = 16 = 4 \times 4$. The inputs to two $\mathbf{DFT}_4$s are emphasized.

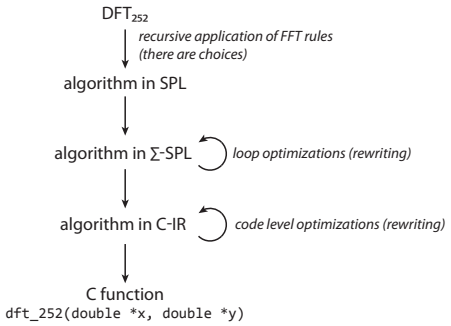

Figure 2. The version of Spiral considered in this paper.

ates unvectorized single-threaded DFT code for arbitrary but fixed input sizes as explained in [12, 27, 42].

Discrete Fourier transform. The DFT multiplies a given complex input vector x of length n by the fixed $n \times n$ DFT matrix to produce the complex output vector y. Formally,

$$y = \mathbf{DFT}_n\, x, \quad \text{where } \mathbf{DFT}_n = [\omega_n^{k\ell}]_{0 \le k,\ell \le n}$$

and $\omega_n = \exp -2\pi\sqrt{-1}/n$.

Fast Fourier transforms (FFTs). Divide-and-conquer FFTs (the algorithm knowledge) in Spiral are represented as rules that decompose $\mathbf{DFT}_n$ into a product of structured sparse matrices that include smaller DFTs. For example, the Cooley-Tukey FFT is given by

$$\mathbf{DFT}_n \to (\mathbf{DFT}_k \otimes I_m) T_m^n (I_k \otimes \mathbf{DFT}_m) L_k^n, \quad n = km, \quad (1)$$

where I_n is the identity matrix, L_k^n is a certain permutation matrix, T_m^n is the diagonal matrix of twiddle factors, and

$$A \otimes B = [a_{k,\ell} B]_{0 \le k,\ell < n} \quad \text{for } A = [a_{k,\ell}]_{0 \le k,\ell < n}\,.$$

This formalism, called SPL, is a DSL that captures the data flow of computation. For example, (1) for $n = 16 = 4 \times 4$ is shown in Fig. 1; each gray block is a $\mathbf{DFT}_4$ that is again computed recursively using (1).

Other FFT rules in our prototype include the prime factor FFT ($n = km,\ \gcd(k, m) = 1$), and the Rader FFT (n is prime):

$$\mathbf{DFT}_n \to V_{k,m}^{-1} (\mathbf{DFT}_k \otimes I_m)(I_k \otimes \mathbf{DFT}_m) V_{k,m}, \quad (2)$$

$$\mathbf{DFT}_n \to W_n^{-1} (I_1 \oplus \mathbf{DFT}_{p-1}) E_n (I_1 \oplus \mathbf{DFT}_{p-1}) W_n. \quad (3)$$

Here, $V,\ W$ are certain permutation matrices, E_n is diagonal, and $\oplus$ is the block-diagonal composition. Recursive application of FFT rules (1)–(3) yields algorithms for a given $\mathbf{DFT}_n$ and there are many choices in this recursion. All FFTs are terminated with the base rule $\mathbf{DFT}_2 \to F_2 = \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]$.

Loop merging. Fig. 1 suggests a recursive computation in four steps: permutation, followed by a loop over smaller FFTs, followed by scaling, followed by another loop over smaller FFTs. This causes four passes over the data, which is inefficient. A better solution fuses the permutation and scaling steps with the subsequent loops. The permutation then becomes a readdressing in the loop. This merging problem becomes more difficult upon recursion. For example, if all rules (1)–(3) are applied (e.g., for $n = pq$, q prime and $q - 1 = rs$) one may encounter the SPL fragment

$$(I_p \otimes (I_1 \oplus (I_r \otimes \mathbf{DFT}_s) L_r^{rs})\, W_q)\, V_{p,q}.$$

The challenge here is to fuse all three permutations into the innermost loop and to simplify the resulting index expression. In Spiral, this is solved using the DSL Σ-SPL and rewriting [12]. Σ-SPL makes loops and index functions explicit. As a simple example, we consider the fragment $(I_4 \otimes \mathbf{DFT}_4) L_4^{16}$ occurring in Fig. 1. First, it is translated into Σ-SPL, then the permutation is fused into the loop, then the resulting composed index function is simplified. All steps are done by rewriting using rules provided to Spiral:

$$\left(\sum_{j=0}^{3} S(h_{4j,1})\, \mathbf{DFT}_4\, G(h_{4j,1}) \right) \mathrm{perm}(\ell_4^{16}) \quad (4)$$

$$\to \sum_{j=0}^{3} S(h_{4j,1})\, \mathbf{DFT}_4\, G(\ell_4^{16} \circ h_{4j,1}) \quad (5)$$

$$\to \sum_{j=0}^{3} S(h_{4j,1})\, \mathbf{DFT}_4\, G(h_{j,4}). \quad (6)$$

$G(.)$ and $S(.)$ are called gather and scatter and are containers for symbolic index functions that can be manipulated. The sum represents a possible loop, and the loop body is a $\mathbf{DFT}_4$ yet to be further expanded.

Generator. The entire generator is shown in Fig. 2 for some example size ($n = 252$). One of many possible algorithms is generated in SPL, translated to and then optimized in Σ-SPL as explained above, and then translated into a C intermediate language using partial unrolling (namely every DFT below a certain size B encountered in the recursion; we use $B = 16$) that represents the computation DAG. On this DAG, various standard and DFT-specific simplifications are done as explained in [15] (e.g., algebraic simplification, constant normalization and propagation); finally the code is unparsed into C. The entire process is wrapped into a search loop that measures runtime and finds the best recursion using dynamic programming.

2.2 Scala and Lightweight Modular Staging

Multi-stage programming (MSP, or *staging* for short) as established by Taha and Sheard [34] aims to simplify program generator development by expressing the program generator and parts of the generated code in a single program, using the same syntax. Traditional MSP languages like MetaOCaml [8] implement staging by providing syntactic quasi-quotation brackets to explicitly delay the evaluation of (i.e., stage) chosen program expressions.

Contrary to dedicated MSP languages, LMS uses only types to distinguish the computational stages. Expressions of type `Rep[T]` in the first stage yield computations of type `T` in the second stage. Expressions of a plain type `T` in the first stage will be evaluated and become constants in the generated code. The standard Scala type system propagates information about which expressions are staged and thus performs a semi-automatic local binding-time analysis (BTA). Thus, LMS provides some of the benefits of automatic partial evaluation [18] and of manual staging.

Example: Data and traversal abstractions. Consider a Scala implementation of a high-level vector data structure backed by an array:

```
class Vector[T](val data: Array[T]) {
  def foreach(f: T => Unit): Unit = {
    var i = 0; while (i < data.length) { f(data(i)); i += 1 }
}}
```

Given this definition, we can traverse a vector using its foreach method; for example to print its elements:

```
vector foreach { i => println(i) }
```

While convenient, the vector abstraction has non-negligible abstraction overhead (e.g., closure allocation and interference with JVM inlining). To obtain high performance code, we would like to turn this implementation into a code generator, that, when encountering a foreach invocation, will emit a while loop instead. Using LMS, we only need to change the method argument and return types, and the type of the backing array, by adding the Rep type constructor to stage selected parts of the computation:

```
class Vector[T](val data: Rep[Array[T]]) {
  def foreach(f: Rep[T] => Rep[Unit]): Rep[Unit] = {
    var i = 0; while (i < data.length) { f(data(i)); i+=1 }}}
```

The LMS framework provides overloaded variants of many operations (e.g. array access data(i)) that lift those operations to work on Rep types, i.e., staged expressions rather than actual data. This allows us to leave the method body unchanged.

It is important to note the difference between types Rep[A=>B] (a staged function object) and Rep[A]=>Rep[B] (a function on staged values). For example, using the latter in the definition of foreach, ensures that the function parameter is always evaluated and unfolded at staging time.

In addition to the LMS framework, we use the Scala-Virtualized compiler [29], which redefines several core language features as method calls and thus makes them overloadable as well. For example, the code

```
var i = 0; while (i < n) { i = i + 1 }
```

will be desugared as follows:

```
val i = __newVar(0); __while(i < n, { __assign(i, i + 1) })
```

The LMS framework provides methods __newVar, __assign, __while, overloaded to work on staged expressions with Rep types.

The LMS extensible graph IR. Another key difference between LMS and earlier staging approaches is that LMS does not directly generate code in source form but provides instead an extensible intermediate representation (IR). The overall structure is that of a "sea of nodes" dependency graph [10]. For details we refer to [28, 30, 32]; a short recap is provided next.

The framework provides two IR class hierarchies. Expressions are restricted to be atomic and extend Exp[T]:

```
abstract class Exp[T]
case class Const[T](x: T) extends Exp[T]
case class Sym[T](n: Int) extends Exp[T]
```

Composite IR nodes extend Def[T]. Custom nodes typically are composite. They refer to other IR nodes only via symbols. There is also a class Block[T] to define nested blocks.

As a small example, we present a definition of staged arithmetic on doubles (taken from [30]). We first define a pure interface in trait Arith by extending the LMS trait Base, which defines Rep[T] as an abstract type constructor.

```
trait Arith extends Base {
  def infix_+(x: Rep[Double], y: Rep[Double]): Rep[Double]
  def infix_-(x: Rep[Double], y: Rep[Double]): Rep[Double]
}
```

We continue by adding an implementation component ArithExp, which defines concrete Def[Double] subclasses for plus and minus operations.

```
trait ArithExp extends BaseExp with Arith {
  case class Plus(x: Exp[Double], y: Exp[Double])
    extends Def[Double]
  case class Minus(x: Exp[Double], y: Exp[Double])
    extends Def[Double]
  def infix_+(x: Exp[Double], y: Exp[Double]) = Plus(x,y)
  def infix_-(x: Exp[Double], y: Exp[Double]) = Minus(x,y) }
```

Trait BaseExp defines Rep[T]=Exp[T], whereas Rep[T] was left abstract in trait Base.

Taking a closer look at ArithExp reveals that the expected return type of infix_+ is Exp[Double] but the result value Plus(x,y) is of type Def[Double]. This conversion is performed implicitly by LMS using toAtom:

```
implicit def toAtom[T](d: Def[T]): Exp[T] = reflectPure(d)
```

The method reflectPure maintains the correct evaluation order by binding the argument d to a fresh symbol (on the fly conversion to administrative normal form (ANF)).

```
def reflectPure[T](d: Def[T]): Sym[T]
def reifyBlock[T](b: =>Exp[T]): Block[T]
```

The counterpart reifyBlock (note the by-name argument) collects performed statements into a block object. Additional reflect methods exist to mark IR nodes with various kinds of side effects (see [32] for details).

3. Implementing the Spiral Prototype Using LMS

In this section we explain the implementation of the generator, as outlined in Section 2.1, in the LMS framework. The section is organized according to the approach presented in Section 1; all of the steps are relevant for the chosen subset of Spiral. The running example will be $\mathbf{DFT}_4$ decomposed using (1):

$$\mathbf{DFT}_4 \rightarrow (\mathbf{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \mathbf{DFT}_2) L_2^4 \qquad (7)$$

3.1 Algorithmic Knowledge as Multiple Levels of DSLs

Spiral requires three DSLs: SPL, Σ-SPL, and an internal representation of C (C-IR); see Fig. 2. We focus on SPL.

DSL representation. The DSL SPL is defined inside Scala in two steps. First, basic matrices such as T_m^n, L_k^n, or $\mathbf{DFT}_2$ are defined as regular Scala classes:

```
abstract class SPL
case class T(n: Int, m: Int) extends SPL
case class DFT(n: Int) extends SPL
case class F2() extends SPL
case class I(n: Int) extends SPL
case class L(n: Int, k: Int) extends SPL
```

Then, matrix operations like product (composition) or $\otimes$ are defined using LMS. The common practice in LMS is to first provide the language interface in terms of abstract methods that operate on (staged) Rep types:

```
trait SPL_Base extends Base {
  implicit def SPLtoRep(i: SPL): Rep[SPL]
  def infix_tensor (x: Rep[SPL], y: Rep[SPL]): Rep[SPL]
  def infix_compose(x: Rep[SPL], y: Rep[SPL]): Rep[SPL] }
```

The method SPLtoRep defines an implicit lifting of SPL operands to Rep[SPL] expressions, and the methods infix_tensor as well as infix_compose define the corresponding operations. Similar to the example in Section 2, we continue with the concrete implementation in terms of the LMS expression hierarchy.

```
trait SPL_Exp extends SPL_Base with BaseExp {
  implicit def SPLtoRep(i: SPL) = Const(i)
  case class Tensor (x:Exp[SPL], y:Exp[SPL]) extends Def[SPL]
  case class Compose(x:Exp[SPL], y:Exp[SPL]) extends Def[SPL]
  def infix_tensor  (x:Exp[SPL], y:Exp[SPL]) = Tensor (x, y)
  def infix_compose (x:Exp[SPL], y:Exp[SPL]) = Compose(x, y) }
```

SPLtoRep instructs the compiler to convert objects of type SPL to their staged version, whenever a compose or tensor operation is applied.

Decomposition and search. As explained in Section 2.1, FFTs are expressed as decomposition rules in SPL. We represent such a rule (e.g., (1)), using Scala's first-class pattern matching expression called *partial function*. The type in our case is

```
type Rule = PartialFunction[SPL,Rep[SPL]]
```

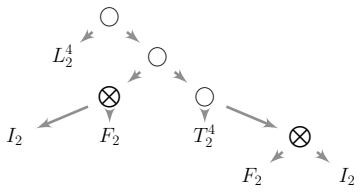

Figure 3. SPL IR representation of a staged $\mathbf{DFT}_4$ decomposition

SPL exp. S	Pseudo code for $y = Sx$	Function
$A_n B_n$	`<code: t = Bx>` `<code: y = At>`	f_{comp}

$I_n \otimes A_n$	`for (i=0; i<m; i++)`
	`  <code: y[i*n:1:i*n+n-1] =`
	`  A(x[i*n:1:i*n+n-1])>`
		f_{ItensorA}

$A_n \otimes I_n$	`for (i=0; i<n; i++)`
	`  <code: y[i:n:i+m*n-n] =`
	`  A(x[i:n:i+m*n-n])>`
		f_{AtensorI}

F_2	`y[0] = x[0] + x[1];` `y[1] = x[0] - x[1];`	f_{F2}

Table 1. SPL to code mapping and name of the emitted functions.

where `SPL` and `Rep[SPL]` are the types of the lefthand side and right-hand side of a rule like (1), respectively. The complete definition of (1) takes the following form. Note how the partial function captures the condition of applicability.

```
val DFT_CT: Rule = {
  case DFT(n) if n > 2 && !isPrimeInt(n) =>
    val (m,k) = factorize(n)
    (DFT(k) tensor I(m)) compose T(n,m)
      compose (I(k) tensor DFT(m)) compose L(n,k)
}
```

In the same fashion we represent a base rule to terminate the algorithm:

```
val DFT_Base: Rule =
  case DFT(2) => F2()
```

Partial functions provide a method `isDefinedAt` that matches an input against the pattern inside the function and returns true if the match succeeds. Hence, we obtain a list of all rules applicable to $\mathbf{DFT}_4$ as follows:

```
val allRules = List(DFT_CT, DFT_Base, ...........)
val applicableRules = allRules filter (_.isDefinedAt(DFT(4)))
```

Partial functions also include an `apply` method that returns the result of the body of the function. Using this method, all algorithms for a $\mathbf{DFT}_n$ can easily be generated. In practice, we utilize a feedback driven dynamic programming search to explore only a subspace of all possible decompositions. In our running example, there is only one algorithm shown in Fig. 3. The circles refer to the `Compose` class, the $\otimes$ to the `Tensor` class; all the leaves are subclasses of `SPL`. This representation can now be transformed using rewriting (see Section 3.2 later), or unparsed into the target language.

Translation. Since we need to further manipulate the generated algorithm, we do not unparse directly to target code. Rather we define a denotational interpretation of the DSL, which maps every node of the IR graph to its "meaning": a Scala function that performs the corresponding matrix-vector multiplication. The in- and output types are arrays of complex numbers. This function can immediately be used to execute the program when prototyping or debugging. In the next section we will derive translations to lower-level DSLs from the interpretation. Examples of these functions are shown in Table 1. Conceptually, they correspond to the templates used in the original SPL compiler [42].

To implement this mapping in Scala, we define an abstract method `transform` in the base class SPL:

```
abstract class SPL {
  def transform(in: Array[Complex]): Array[Complex] }
```

and provide implementations for each concrete subclass (e.g., mapping F_2 to f_{F2}).

```
case class F2() extends SPL {
  override def transform(in: Array[Complex]) = {
    val out = new Array[Complex](2)
    out(0) = in(0) + in(1)
    out(1) = in(0) - in(1)
    out   }}
```

The definition of complex numbers is straightforward.

```
case class Complex(_re: Double, _im: Double) {
  def plus(x: Complex, y: Complex)
    = Complex(x._re + y._re, x._im + y._im)
  def minus(x: Complex, y: Complex)
    = Complex(x._re - y._re, x._im - y._im)  }
```

In addition to the SPL operands, we need to translate the tensor and compose operations. We provide suitable functions for each individual case, for example

```
def I_tensor_A(I_n: Int, A: (Array[Complex]=>Array[Complex]))
  = { in: Array[Complex] =>
        in.grouped(in.size/I_n) flatMap (part => A(part))  }
```

To obtain an interpretation of a given SPL program, we traverse the SPL IR graph (e.g., Fig. 3) in dependency order, call for every node the appropriately parameterized function:

```
def translate(e: Exp[SPL]) = e match {
  case Def(Tensor(Const(I(n)), Const(a: SPL))) =>
    I_tensor_A(n, a.transform)
    .....
}
```

The pattern extractor `Def` is provided by LMS and will look up the right-hand side definition of an expression in the dependency graph. The result of invoking `translate` on the topmost node in the SPL IR yields the desired DFT computation as a Scala function of type `Array[Complex]=>Array[Complex]`. In the running $\mathbf{DFT}_4$ example, the generated call graph takes the following form:

$$f_{\text{comp}}(f_{\text{comp}}(f_{\text{comp}}(f_{\text{AtensorI}}(f_{\text{F2}}, f_{\text{I}}), f_{\text{T}}), f_{\text{ItensorA}}(f_{\text{I}}, f_{\text{F2}})), f_{\text{L}}) \quad (8)$$

In summary, at this stage we have already constructed an internal DSL, which can be used within the native environment of Scala.

Translation to another DSL. Running an internal DSL in a library fashion is convenient for debugging and testing. However, for the generator we need to be able to translate one DSL into another DSL, to rewrite on the DSL, and to unparse the DSL into a chosen target language. Next, we show how to translate SPL into another DSL: an internal representation of a subset of C, called C-IR, for further optimization. We omit the step through Σ-SPL shown in Fig. 2 due to space limitations, but the technique used for translation is analogous.

To translate to C-IR, only a very minor change is required: the parameters of the class `Complex` are annotated with `Rep` for staging:

```
case class Complex(_re: CIR.Rep[Double], _im: CIR.Rep[Double])
```

Note that since we are now working with multiple DSLs, we need to specify which language we are referring to by using `SPL.Rep` or `CIR.Rep`. In unambiguous cases we omit the prefix and leave it to Scala's scoping mechanism. Invoking `translate` as defined above now yields a function that returns an IR representation of the computation, instead of the actual computation result. Enveloping the generated function within a wrapper as shown below yields the C-IR representation depicted in Fig. 4.

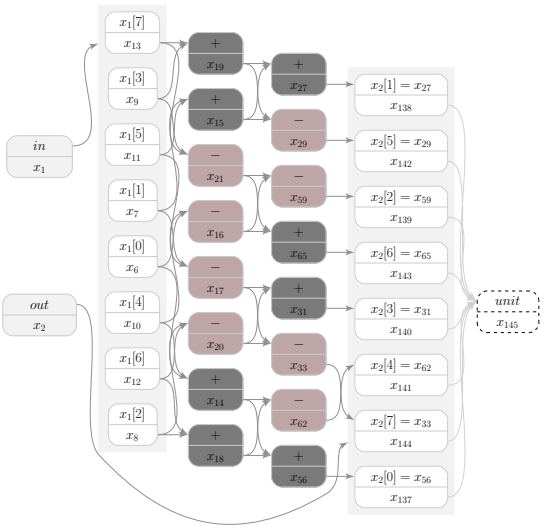

Figure 4. C-IR representation of a staged $\mathbf{DFT}_4$ decomposition for complex input.

```
def wrapper(f: Array[Complex]=>Array[Complex], dft_size: Int)
     (in: Rep[Array[Double]], out: Rep[Array[Double]]) = {
  val scalarized = new Array[Complex](dft_size)
  for (i <- 0 until dft_size)
    scalarized(i) = Complex(in(i*2),in(i*2+1))
  val res = f(scalarized)
  for (i <- 0 until dft_size) {
    out(i*2) = res(i)._re
    out(i*2+1) = res(i)._im
}}
```

This wrapper transforms a staged double array into another staged array by calling the created function `f` within the snippet. Note that the implementation commits to a specific encoding of complex arrays into double arrays (as shown: interleaved format). We will abstract over the choice of representation in Section 3.3.2. The `wrapper` also commits to a specific code style, namely unrolling with scalar replacement; abstraction over this choice is also explained later. The white boxes in Fig. 4 correspond to the reads of and writes to the staged array; the other boxes are arithmetic operations on staged doubles. Note how any abstraction overhead in the function callgraph is gone due to unrolling and only the operations on the staged variables `_re` and `_im` remain. Unparsing to actual C code is now straightforward.

3.2 Optimization through DSL Rewriting

Most domain-specific optimizations in Spiral are done by rewriting DSL expressions. In case of our prototype this occurs on two levels (Fig. 2): Σ-SPL and C-IR. LMS provides rewriting support through its transformer infrastructure [28]. Combined with the pattern matching support of Scala, the rewrite rule used in (5), for example, takes the following form:

```
override def transformStm(stm: Stm):Exp[Any]= stm.rhs match {
  case Compose(Const(g: Gather), Const(lperm: L)) =>
    Gather(compose(l(lperm.k,lperm.m),g.f))
  case _ => super.transformStm(stm)
}
```

The same infrastructure is used to optimize the C-IR graph. For example, the simplification of multiplications by 0 or 1 and constant folding are implemented as follows:

```
override def transformStm(stm: Stm):Exp[Any]= stm.rhs match {
  case NumericTimes(a, b) => (this(a), this(b)) match {
    case (Const(p), Const(q)) => Const(p * q)
    case (_, Const(0)) | (Const(0), _) => Const(0)
    case (e, Const(1)) | (Const(1), e) => e
    case (e1, e2) => e1 * e2
  }
  case _ => super.transformStm(stm) }
```

The operation `this(a),this(b)` applies the enclosing transformer object to the arguments `a,b` of the multiplication statement. The optimizations implemented in our prototype include common subexpression elimination, constant normalization, DAG transposition and others from [15].

Scala provides additional pattern matching flexibility with custom *extractor objects*. Any object that defines a method `unapply` can be used as a pattern in a match expression. An example were we use this are binary rewrite rules over longer sequences of expressions. Consider for example a putative simplification rule $H(m) \circ H(n) \to H(m+n)$. We would like to apply this rule to a sequence of $\circ$ operations, such that for example $A \circ H(m) \circ H(n) \circ B$ becomes $A \circ H(m+n) \circ B$. This can be achieved in two steps. We use type `IM` as a shortcut for `Rep[SPL]`. First, we define a custom extractor object:

```
object First {
  def unapply(x: IM): Option[(IM, IM=>IM)] = x match {
    case Def(Compose(a,b)) => Some((a, (a1 => a1 compose b)))
    case _ => Some((x, x1 => x1))  }}
```

Matching `First` against $A \circ B$ will extract (A, w) where w is a function that replaces A, i.e., $w(C) = C \circ B$. Matching against just A will return (A, id).

In the second step, we define a "smart" constructor for the $\circ$ operation `Compose` that uses `First` to generalize the binary rewrite:

```
def infix_compose(x: IM, y: IM): IM = (x,y) match {
  case (Const(H(m)), First(Const(H(n)), wrap)) =>
    wrap(H(m+n))
  case (Def(Compose(a,b)),c) =>
    a compose (b compose c)
  case _ =>
    Compose(x,y)
}
```

If the rewrite is not directly applicable, another case is tried that will canonicalize $(A \circ B) \circ C$ to $A \circ (B \circ C)$. Only if that fails, an IR node `Compose(a,b)` is created. Finally, a transformer needs to be created that invokes the smart constructor:

```
override def transformStm(stm: Stm):Exp[Any]= stm.rhs match {
  case Compose(x,y) => this(x) compose this(y)
  case _ => super.transformStm(stm)}
```

In our generator we use this feature to implement most of the Σ-SPL rewrites including those sketched in (5) and (6).

3.3 Abstracting Over Data Representations and Code Style

In this section we discuss techniques to abstract over data layouts that go hand in hand with performance transformations such as selective precomputation, unrolling with scalar replacement, and specialization. The key insight is to abstract over staging decisions: we will be able to generate data structures or code patterns where different pieces are evaluated at generation time or computed by the generated code, depending on particular type instantiations.

We will make use of the type class design pattern [25, 40], which decouples data objects from generic dispatch and thus combines naturally with a staged programming model. As an example that we use later, we define a variant of Scala's standard `Numeric` type class that enables abstraction over different numeric types including double, float, and complex:

```
trait NType[T] {
  def plus (x: T, y: T): T
  def minus(x: T, y: T): T }
```

It is easy to define an instance for numeric operations on doubles:

```
implicit object doubleNT extends NType[Double] {
  def plus (x: Double, y: Double) = x + y
  def minus(x: Double, y: Double) = x - y }
```

As an example of using the generic types, we extend our earlier definition of complex numbers to abstract over the component type:

```
case class Complex[T:NType](_re: T, _im: T) {
  def num = implicitly[NType[T]]
  def +(that: Complex) =
    Complex(num.plus(_re, that._re), num.plus(_im, that._im))
  def -(that: Complex) = ... }
```

We use Scala's `implicitly` operation to access the type class instance that implements the actual `plus` and `minus` operations. Type classes in Scala are implemented as implicit method parameters. Thus, the above class definition could equivalently be written as

```
case class Complex[T](_re: T, _im: T)(implicit num: NType[T])
```

Now that we have defined complex numbers, we can turn them into numeric objects as well:

```
implicit def complexNT[T:NType] extends NType[Complex[T]] {
  def plus(x: Complex[T], y: Complex[T]) = x + y
  def minus(x: Complex[T], y: Complex[T]) = x - y }
```

To make the generation of C-IR as flexible as possible, we employ type classes to abstract over the choice of numeric types. In our context this means changing the signatures of our `transform` methods on `SPL` objects to the following format:

```
def transform[T:NType](in: Array[T]): Array[T] = ...
```

3.3.1 Selective Precomputation

Precomputation is naturally supported by a staging framework such as LMS. Fine grain control over which parts should be precomputed is possible by using `Rep` types in suitable places. In many cases it is desirable to abstract over this decision, which is done using type classes as explained next. Afterwards we show as example the selective precomputation of the twiddle factors (the constants in T^n_m) in (1).

Selective staging. To abstract over the staging decision in addition to abstracting over the numeric data type as explained above, we define `NType` instances for each numeric type. For example, for doubles it becomes

```
implicit object doubleRepNT extends NType[Rep[Double]] {
  def plus(x: Rep[Double], y: Rep[Double]) = x + y
  def minus(x: Rep[Double], y: Rep[Double]) = x - y }
```

Using this mechanism, we can turn staging on or off by providing the corresponding type when calling the transform function. For example, we can now invoke the same definition of `transform` with any of the following types:

```
Array[Double]          Array[Complex[Double]]
Array[Rep[Double]]     Array[Complex[Rep[Double]]]
```

The same mechanism enables further powerful abstractions that are explained below. In particular, the abstraction over the choice between interleaved and split complex format and over the choice between scalar replacement and array computations.

Precomputation. Precomputation is a classic performance optimization. An example in the context of the FFT are the constant twiddle factors required during the Cooley-Tukey FFT (1). Those numbers require expensive sin and cos operations. It usually pays off to precompute those numbers and inline them as constants in the code or store them in a table. For very large sizes, when the FFT becomes memory-bound, a computation on the fly may be preferrable. Using selective staging we can abstract over this decision by simply instantiating the twiddle computation with a suitable type. The generic computation for one twiddle factor is

```
case class E[T:NType](val n : Int, val k : Int) {
  def re(p: T): T = cos(2.0 * math.Pi * p * k, n)
  def im(p: T): T = sin(2.0 * math.Pi * p * k, n) * -1.0 }
```

Instantiating with `Double` or `Rep[Double]` controls the precomputation. The latter needs staged sin and cos implementations.

3.3.2 Abstraction over Data Representations

One of the cumbersome programming tasks in the creation of a program generator is support for different data layouts. In our case of FFTs, this would be different ways of storing complex numbers, including as interleaved, split, and C99 complex arrays. In this section, we explain how to abstract over this choice.

So far we have been using plain arrays to hold input, intermediate, and output data. To abstract over the data representation, we first define a new, abstract collection class `Vector` with an interface similar to arrays:

```
abstract class Vector[AR[_], ER[_], T] {
  def apply(i: AR[Int]): ER[T]
  def create(size: AR[Int]): Vector[AR, ER, T]
  def update(i: AR[Int], y: ER[T])
}
```

In contrast to arrays, however, `Vector` is parametric in two type constructors: `AR` and `ER`. The type constructor `AR` (short for AccessRep) wraps the indices that are used to access elements, and `ER` (short for ElemRep) wraps the type of elements. Instantiating either or both of these type constructors as `Rep` or `NoRep` (with `NoRep[T]=T`) will yield a data structure with different aspects staged. Moreover, `ER` can be instantiated to `Complex` to explicitly model vectors of complex numbers.

We also want to implement subclasses of `Vector` that abstract not only over the data layout but also over the choice of staging the internal storage or not (this is equivalent to scalarization of arrays discussed later). To do this we introduce another abstraction of arrays, which is less general, and only wraps a single type constructor `AR` around all operations:

```
trait ArrayOps[AR[_], T] {
  def alloc(s: AR[Int]): AR[Array[T]]
  def apply(x: AR[Array[T]], i: AR[Int]): AR[T]
  def update(x: AR[Array[T]], i: AR[Int], y: AR[T])
}
```

Instances of `ArrayOps` will be used as type class arguments by `Vector` subclasses to abstract over plain and staged internal arrays (i.e., `AR=Rep` or `NoRep`).

Finally we have all constructs to represent a variety of different data layouts. We demonstrate with split complex (real and imaginary parts in separate arrays) and C99 complex arrays:

```
class SplitComplexVector[AR[_], T:NType](size: AR[Int])
  (implicit aops: ArrayOps[AR, T])
  extends Vector[AR, Complex, AR[T]] {
    val dataRe: AR[Array[T]] = aops.alloc(size)
    val dataIm: AR[Array[T]] = aops.alloc(size)
    def create(size: AR[Int]) = new SplitComplexVector(size)
    def apply(i: AR[Int]): Complex[AR[T]] =
      new Complex(_re = aops.apply(dataRe, i),
                  _im = aops.apply(dataIm, i))
    def update(i: AR[Int], y: Complex[AR[T]]) = {
      aops.update(dataRe,i,y._re)
      aops.update(dataIm,i,y._im)
    }}
class C99Vector[AR[_],T:NType](s: AR[Int])
  (implicit aops: ArrayOps[AR, Complex[T]])
  extends Vector[AR, AR, Complex[T]] {
    val data = aops.alloc(s)
    def create(size: AR[Int]) = new C99Vector[AR,T](size)
    def apply(i: AR[Int]): AR[Complex[T]] = aops.apply(data,i)
    def update(i: AR[Int], y: AR[Complex[T]])
      = aops.update(data,i,y)
}
```

The split complex implementation abstracts over staging via the type constructor parameter AR and contains elements of type Complex[AR[T]]. Thus, it extends Vector[AR,Complex,AR[T]]. An implementation of interleaved storage using a single array would use the same type. In contrast, the variant that implements arrays of C99 complex numbers specifies its element type as AR[Complex[T]] and therefore extends Vector[AR,AR,Complex[T]] The vector classes manage either one or two backing arrays using the operations of the aops type class instance, which is passed as an implicit constructor parameter. The accessor methods apply and update map element from the internal data arrays to an external interface and vice versa. In the split complex case, the external representation is always a staging-time Complex object.

Generalizing the generating functions. To accommodate the new generalized data structures, we have to slightly extend the interface of the transform method that emit the staged C-IR:

```
case class F2() extends SPL {
  override def transform[AR[_],ER[_],T:NType]
      (in: Vector[AR,ER,T]) = {
    val out = in.create(2)
    out(0) = in(0) + in(1)
    out(1) = in(0) - in(1)
    out
}}
```

Calling this generalized F2 function with the input

```
val in = new SplitComplexVector[Rep, Double](2)
```

will be resolved as

```
transform[Rep,Complex,Double](in: Vector[Rep,Complex,Double])
```

In other words, the internal storage type will be Rep[Array[Double]]. Therefore, array operations will appear in the resulting C-IR graph. The complex class, which is mainly used to enable more concise code, does not occur in the staged IR, therefore not causing any overhead.

In addition to the staged array data representations, we can also create a scalarized version:

```
val in = new SplitComplexVector[NoRep,Rep[Double]](2)
```

In this version, the array becomes a regular Scala array that contains staged values (Array[Rep[Double]]). The resulting C-IR graph does not contain any of the array or complex operations performed at staging time.

3.3.3 Unrolling and Scalar Replacement

We explain how to abstract over the code style.

Looped code. Beside variables and their operations, also control structures such as loops, conditionals and functions can be staged, as briefly shown already in section 2.2. For the I_tensor_A function introduced in section 3.1, extended by the abstractions introduced in 3.3.2, looped code can be implemented as follows:

```
def I_tensor_A[AR[_], ER[_], T:NType](size: Int, n: Int,
    A: Vector[AR,ER,T] => Vector[AR,ER,T]) = {
  in: Vector[AR,ER,T] =>
  val out = in.create(size)
  val n_staged: Rep[Int] = n
  val frag: Rep[Int] = size/n
  for (i <- 0 until n_staged) {
    val tmp = in.create(frag)
    for(j <- 0 until frag) tmp(j) = in(i*n+j)
    val t = A(tmp)
    for(j <- 0 until frag) out(i*n+j) = t(j)
  }
  out  }
```

Note that the variables n_staged and frag are annotated with a Rep type, therefore causing the for loop expression to be staged.

Scalarization. In mathematical high performance code, unrolling and scalar replacement in static single assigment (SSA) form is a standard optimization. It explicitly copies array elements that are reused into temporary variables and removes false dependencies; this way, the compiler is able to rule out memory aliasing and thus to perform better register allocation and instruction scheduling. Scalarization and SSA form come very naturally with LMS as already shown in Fig. 4. By moving the data from a staged array into a Scala container-object containing single staged variables, scalarization effectively takes place. For every operation result gained from this variables, LMS creates a new variable, thus producing SSA form. Using the constructs from Section 3.3.2, scalarization is done by simply moving data between containers:

```
val staged_array: SplitComplexVector[Rep,Double]
val scalarized=new SplitComplexVector[NoRep,Rep[Double]](size)
for (i <- until size)  scalarized(i) = staged_array(i)
for (j <- until size)  SomeComputation(scalarized(j))
for (i <- until size)  staged_array(i) = scalarized(i)
```

The value size is a non-staged integer. Next, we combine scalarization with unrolling.

Unrolling. To perform partial unrolling to enable scalarization, we just need to combine the concepts we have seen so far. In particular we scalarize at the beginning of the code fragment we want to unroll, replacing the staged loops with regular Scala loops.

```
def I_tensor_A[AR[_],ER[_],T:NType](size: Int, n:Int,
    A: Vector[AR,ER,T] => Vector[AR,ER,T]) = {
  in: Vector[AR,ER,T] =>
  val in_scalar = new SplitComplexVector[NoRep,ER[T]](size)
  val out = in.create(size)
  val frag = size/n
  for (i <- 0 until size) in_scalar(i) = in(i) //scalarize
  for (i <- 0 until n) {  //start unrolling
    val tmp = in.create(frag)
    for(j <- 0 until frag) tmp(j) = in(i*n+j)
    val t = A(tmp)
    for(j <- 0 until frag) out(i*n+j) = t(j)
  } //end of unrolling
  out  }
```

Instead of manually implementing scalarization for each loop, we can also introduce higher level methods to perform this conversion.

3.3.4 Specialization

Specialization is another important performance optimization and an ability that can distinguish library generators from manually written libraries. In the case of FFT, relevant opportunities for specialization include the presence of symmetry in the input (e.g., the second half is a mirrored version of the first half) or fixing certain inputs (e.g., all imaginary parts are zero). In both cases operations can be reduced. In LMS and our prototype, many specialization cases can be expressed by function composition, where the inner, more general function is wrapped into an outer one that replaces parts of the generic input to make specialization patterns explicit. To illustrate, we assume a function f on complex arrays.

```
val f: Array[Complex] => Array[Complex] = ...
```

If we want to specialize f to an input whose first element is known to be 1.0, a specialized version f_spec with the same signature can be obtained as follows (in.copy hides the assignment from the caller):

```
val f: Array[Complex] => Array[Complex] = ...
val f_spec = { in: Array[Complex] =>
  val in_spec = in.copy()
  in_spec(0) = Complex(1.0, 0.0)
  f(in_spec)}
```

This pattern is detected at the C-IR optimization level and will result in the elimination of each multiplication with the real part, and each addition and subtraction with the imaginary part. Further dead code removal and opportunities for other simplifications may apply.

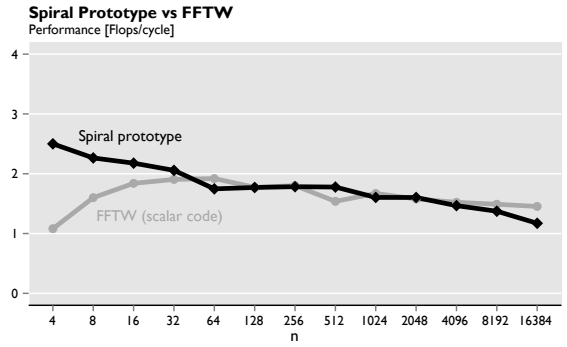

Figure 5. Our generated code versus FFTW 3.3.2

In general, in a specialization the IR graph will be pruned, leading to a simplified version of the initial graph with less operations and thus better performance. In our prototype, the specialization cases mentioned above are supported for scalarized code. This is also the case in FFTW codegen [15].

4. Experimental Results

We compare the performance of the code generated by our Spiral prototype (Fig. 2) against the performance of the carefully hand-written and hand-optimized FFTW 3.3.3 (only the base cases in FFTW are (pre)generated), known to be one of the fastest libraries available. We keep the evaluation brief since the main contribution of this paper is in the demonstration of how to use program language features to build generators.

Experimental setup. The experiments were performed on an Intel i7-2620M with Ubuntu 11.10, using icc 13.0 with flags -O3 -no-simd -no-vec -xHost. The timing is the minimum of ten repeated warm cache measurements with the TSC hardware performance counter. The code considered for both FFTW and our prototype is scalar code (no SSE/AVX vectorization) without threading and with in- and output in interleaved format. FFTW was used with its search enabled. The prototype uses a dynamic programming search and unrolls once the recursion reaches a transform of size ≤ 32. The entire code generation time was less then an hour.

Discussion. Fig. 5 shows the pseudo performance using the pseudo-flop count of $5n \log_2(n)$ on increasing two power input sizes. The plot demonstrates that our prototype yields performance comparable to the one of FFTW. The development of the generator took about 15 man months by two people with no prior experience of Spiral, Scala, or staging.

5. Related Work

There is a considerable body of work on individual program generators (as surveyed in Section 1), but systematic work on implementation methodologies for high-performance program generation is far less widespread.

The original FFTW codelet generator [15] was implemented in OCaml, whose functional programming features such as pattern matching are a good fit for symbolic manipulation. However, a key element of the FFTW simplifier is to provide a tree-like interface to an internal DAG representation. This is achieved by a monadic front-end layer, which also eliminates common subexpressions using memoization. As a consequence, the simplifier needs to be written in explicit monadic style, which adds some notational overhead.

Lisp and Scheme have for a long time supported variants of quote/unquote to compose program fragments (*quasi-quotation*). Racket [35] is a modern dialect with powerful macro facilities. However it is not clear whether sophisticated abstractions (e.g. over data layout) can be as easily achieved without a strong static type system (type classes, etc). In a statically typed setting, language support for quasi-quotation was introduced by MetaML [34] and MetaOCaml [8]. Much of the research around these multi-stage languages focuses on extended static guarantees, such as well-scoping and well-typing of generated code. The core abstraction remains an essentially syntactic expansion facility: Composed code fragments cannot be inspected or further transformed. Thus, MetaOCaml encourages a *purely* generative approach which rules out multiple levels of DSLs. MetaOCaml has been used to develop the FFT codelets needed by FFTW [21, 22] but most of the work is performed by tailor-made front-end layers that implement custom abstract interpretations, not the staging facilities themselves. Cohen et. al. [11] demonstrate how a range of loop transformations can be implemented in a purely generative setting, but they also note the limitations, namely when it comes to composing sequences of transformations.

On the opposite end of the spectrum, there are purely transformational systems. Examples are language workbenches such as JetBrains MPS [17] or Spoofax [20] and rewriting engines such as Stratego/XT [6]. While these systems make it easy to compose and layer transformations, it takes additional steps to execute arbitrary code during a transformation. For example, using numeric libraries, or storing pieces of intermediate code in a hash table is not as straightforward as in a purely generative approach.

We believe that successful environments will most likely not be found at the extremes of the spectrum but will offer well-chosen compromises. LMS in particular provides safety assurances for common uses, but also offers an extensible IR with transformation and rewriting support. LMS is a core component of the Delite DSL framework [7, 23, 32], which has been used to implement high-performance DSLs such as OptiML [33].

Limited forms of program generation can be achieved using C++ expression templates [36]. Examples are libraries such as Blitz++ [37], POOMA[19] or Eigen [1], which implement varying degrees of optimizations. However, expressing transformations in the template language can be awkward, and there is no support for non-local transforms that operate across different template expressions or calling library functions at generation time.

Finally, the original Spiral [26] system was implemented inside the computer algebra system GAP for group theory and abstract algebra. GAP offers a rich set of transform-relevant functionality but not much beyond. Most of the required features (DSLs, rewriting, transformations) where thus implemented by extending the environment and without particular language support.

6. Conclusions

Traditionally, the community that aims for highest performance code with detailed architectural- and microarchitectural-cognizant optimizations and the community that builds programming languages and tools are somewhat separated. In the last decade, the difficulty of optimization has led the former to slowly start using DSLs and program generation; however, the implementations usually don't leverage advanced programming techniques and environments. The main goal of this paper was to show that using the proper techniques, such generators can be constructed in a more principled, systematic way with results that are easier to maintain and extend.

In our case study, a small Spiral prototype implemented in Scala using LMS, we demonstrated how to solve several key challenges involved in building generators within a staging framework. Of particular interest are the translation between multiple levels of DSLs, and the abstraction over data representations and different performance code styles with configurable types. This includes unrolling with scalar replacement, a widespread necessary but ugly transformation. The downside is the increased level of expertise required by the programmer to understand and properly use the advanced con-

cepts. For the case study, which was started without prior Scala experience, this meant several reimplementations yielding more and more concise code and eventually the solutions presented in this paper. However, for an artifact as complex as a program generator, we believe this is a price worth paying.

Scala with LMS was one of many choices in consideration at the start of the project. Besides the obvious benefits of having the expertise of the co-authors available by choosing Scala with LMS, also other considerations drove the decision. In particular, experience with Spiral has shown that frequently auxiliary functionality is needed in building a generator; thus, the interoperability of Scala with Java, and thus access to existing Java libraries was a major advantage. In the case study, for example, we used JTransforms to verify generated code and to precompute E_n in (3) and many more libraries such as the Apache Math Commons, the Scala Language-Integrated Connection Kit, ScalaTest, Scalaz trees and Bridj. For this reason, we did not consider tool chains that provide external DSLs. We benefited from other language features in Scala; for example, we used the object-oriented paradigm to structure our implementation and to represent our DSLs, and we used the functional paradigm to express the mathematical algorithms that derive the generated code. For rewriting we benefited from the support for pattern matching and extractors. Finally, we also took into consideration the long term support of all environments in consideration, a relevant issue for long running projects such as Spiral.

References

[1] Eigen C++ template library for linear algebra. `http://eigen.tuxfamily.org`.

[2] G. N. W. A. Logg, K.-A. Mardal, editor. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[3] B. Aktemur, Y. Kameyama, O. Kiselyov, and C.-c. Shan. Shonan challenge for generative programming: short position paper. In *Proc. Partial evaluation and program manipulation (PEPM)*, pages 147–154, 2013.

[4] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek. Automating the generation of composed linear algebra kernels. In *SC*. ACM, 2009.

[5] J. Bilmes, K. Asanović, C. whye Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proc. Int'l Conference on Supercomputing (ICS)*, pages 340–347, 1997.

[6] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, 2008.

[7] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proc. Parallel Architectures and Compilation Techniques (PACT)*, pages 89–100, 2011.

[8] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, Gensym, and reflection. In *Proc. Generative Programming and Component Engineering (GPCE)*, pages 57–76, 2003.

[9] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *Proc. Int'l conference on object oriented programming systems languages and applications (OOPSLA)*, pages 835–847, 2010.

[10] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17:181–196, March 1995.

[11] A. Cohen, S. Donadio, M. J. Garzarán, C. A. Herrmann, O. Kiselyov, and D. A. Padua. In search of a program generator to implement generic transformations for high-performance computing. *Sci. Comput. Program.*, 62(1):25–46, 2006.

[12] F. Franchetti, Y. Voronenko, and M. Püschel. Formal loop merging for signal transforms. In *Programming Languages Design and Implementation (PLDI)*, pages 315–326, 2005.

[13] F. Franchetti, Y. Voronenko, and M. Püschel. FFT program generation for shared memory: SMP and multicore. In *Supercomputing (SC)*, 2006.

[14] F. Franchetti, Y. Voronenko, and M. Püschel. A rewriting system for the vectorization of signal transforms. In *High Performance Computing for Computational Science (VECPAR)*, volume 4395 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2006.

[15] M. Frigo. A fast Fourier transform compiler. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 169–180, 1999.

[16] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. on Mathematical Software*, 27(4):422–455, 2001.

[17] JetBrains. Meta Programming System, 2009.

[18] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[19] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T. J. Williams. Array design and expression evaluation in POOMA II. In *ISCOPE*, pages 231–238, 1998.

[20] L. C. L. Kats and E. Visser. The Spoofax language workbench. rules for declarative specification of languages and IDEs. In *SPLASH/OOPSLA Companion*, pages 237–238, 2010.

[21] O. Kiselyov, K. N. Swadi, and W. Taha. A methodology for generating verified combinatorial circuits. In G. C. Buttazzo, editor, *EMSOFT*, pages 249–258. ACM, 2004.

[22] O. Kiselyov and W. Taha. Relating FFTW and split-radix. In Z. Wu, C. Chen, M. Guo, and J. Bu, editors, *ICESS*, volume 3605 of *Lecture Notes in Computer Science*, pages 488–493. Springer, 2004.

[23] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.

[24] J. Mattingley and S. Boyd. CVXGEN: A code generator for embedded convex optimization. *Optimization and Engineering*, 13(1):1–27, 2012.

[25] U. Norell and P. Jansson. Polytypic programming in Haskell. In P. W. Trinder, G. Michaelson, and R. Pena, editors, *IFL*, volume 3145 of *Lecture Notes in Computer Science*, pages 168–184. Springer, 2003.

[26] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[27] M. Püschel, B. Singer, M. Veloso, and J. M. F. Moura. Fast automatic generation of DSP algorithms. In *International Conference on Computational Science (ICCS)*, volume 2073 of *Lecture Notes In Computer Science*, pages 97–106. Springer, 2001.

[28] T. Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. PhD thesis, EPFL, 2012.

[29] T. Rompf, N. Amin, A. Moors, P. Haller, and M. Odersky. Scala-virtualized: Linguistic reuse for deep embeddings. In *Higher-Order and Symbolic Computation (Special issue for PEPM'12, to appear)*.

[30] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[31] T. Rompf, A. K. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs. In *Proc. Principles of programming languages (POPL)*, pages 497–510, 2013.

[32] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented DSLs. DSL, 2011.

[33] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, ICML, 2011.

[34] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[35] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Programming language design and implementation (PLDI)*, PLDI '11, pages 132–141, 2011.

[36] T. L. Veldhuizen. Expression templates, C++ gems. SIGS Publications, Inc., New York, NY, 1996.

[37] T. L. Veldhuizen. Arrays in blitz++. In *ISCOPE*, pages 223–230, 1998.

[38] Y. Voronenko, F. de Mesmay, and M. Püschel. Computer generation of general size linear transform libraries. In *International Symposium on Code Generation and Optimization (CGO)*, pages 102–113, 2009.

[39] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC*, volume 16 of *Journal of Physics: Conference Series*, pages 521–530, 2005.

[40] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.

[41] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[42] J. Xiong, J. Johnson, R. W. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *Programming Languages Design and Implementation (PLDI)*, pages 298–308, 2001.